6-2-2011

Constant-RMR Abortable Reader-Priority Reader-Writer Algorithm

Nan Zheng
Dartmouth College

Dartmouth Computer Science Technical Report TR2011-685

Constant-RMR Abortable Reader-Priority Reader-Writer Algorithm

Nan Zheng

Thesis Advisor: Prasad Jayanti

June 2, 2011

Abstract

The concurrent reader-writer problem [6] involves two classes of processes: readers and writers, both of which wish to access a shared resource. Many readers can access the shared resource at the same time. However, if a writer is accessing the resource, no readers or other writers can access the resource at the same time. In the reader-priority version of the problem, readers are prioritized over writers when processes from both classes are trying to access the shared resource. Previous research [2] showed a reader-priority constant-RMR multi-reader, multi-writer algorithm for Cache-Coherent (CC) systems. However, this algorithm does not allow for readers or writers to abort, which allows readers and writers waiting for the resource to stop trying to access the resource and to quickly return to the Remainder Section of the code, where the process performs tasks unrelated to the shared resource.

This thesis presents an abortable constant-RMR reader-priority multi-reader single-writer algorithm for CC systems. Additionally, we show how to generalize the algorithm into a multi-reader multi-writer algorithm using any given abortable mutual exclusion algorithm. The algorithm is proven rigorously by invariants and tested using a system of mathematical specification and model-checking tools (PlusCal/TLA+/TLC).

Contents

1 Introduction

In concurrent systems, computation is performed by multiple processes with a shared memory framework. Processes may perform computations asynchronously, involving both local and shared variables. Processes can execute atomic steps (i.e. a step that whose execution cannot be interrupted by that of another process) of code in any order. Concurrent algorithms define code for individual processes to execute in order to correctly interact with other processes in the system.

One of the central areas of research in concurrent programming is the problem of mutual exclusion[9], where many processes contend for a common shared resource that can only be accessed by one process at a time. A well-studied variation of the mutual exclusion problem is the reader-writer problem, where processes attempting to access the shared resource are divided into two classes: readers and writers. Many readers can access the shared resource at the same time. However, if a writer is accessing the resource, no readers or other writers can access the resource at the same time.

There are a number of ways to specify the problem depending on how one prioritizes the classes accessing the shared resource. The three most common specifications include reader-priority (where readers have priority over writers), writer-priority (where writers have priority over readers), and fair-switching or starvation-free (where neither class has priority over the other, and any process wishing to access the shared resource will eventually get its turn). This thesis studies only the *reader-priority* version of the problem.

Previous research by Bhatt and Jayanti [2] showed a constant-time reader-priority reader-writer algorithm satisfying a large set of desirable properties. However, this algorithm does not allow for reader or writer abort, which allows readers and writers waiting for the resource to stop trying to access the resource and to quickly return to the Remainder Section of the code, where the process performs tasks unrelated to the shared resource.

This thesis presents an *abortable* reader-priority multi-reader single-writer algorithm. The algorithm presented has constant-time reader and writer abort features while preserving constant-time for the rest of the algorithm in hardware systems where caches *are not* invalidated when a variable is updated with the same value. In systems where caches *are* invalidated when a variable is updated to the same value, we show that our algorithm runs in amortized constant-time. We also present

a modification to our algorithm for it to run in constant-time in the latter hardware model for up to R readers (where R is a constant), beyond which the time-complexity will be $O(\frac{n_r}{R})$, where n_r is the number of actual readers in the algorithm.

Additionally, we show how to generalize the algorithm into a multi-reader multi-writer algorithm using any given abortable mutual exclusion algorithm. For example, we can use Jayanti's abortable mutual exclusion algorithm [10] to generate a $O(min(k, log(n_w)))$-time multi-reader multi-writer algorithm, for which k is the number of contending writers for the critical section and n_w is the total number of writers.

Additionally, we provide a formal specification of the single-writer version of the algorithm in PlusCal [13], a formal language that translates pseudocode into TLA+ (Temporal Logic of Actions) [12], a language for specifying and proving properties of both concurrent and sequential systems. We then verify the specifications in the TLC model checker [12] for small sample sets. Finally, we present a formal invariant proof for our algorithm.

2 Background and Definitions

In our computing model, processes communicate asynchronously using shared variables. Each process has its own unique identifier called a process ID (*pid*). We denote the set of all *pid*s as PID. Each process p has a *state*, which is the current value of p's program counter (PC) and all of p's local variables. We denote p's PC as PC_p. The $configuration$ of a the system is defined by the values of all the shared variables and the states of all the processes in the system. The *initial* $configuration$ ($\mathcal{C}_0$) of a system is the configuration before any processes have executed any lines of code.

Processes execute atomic operations by taking *steps*. When it is a process p's turn, p transitions from its current configuration $\mathcal{C}$ by taking a step s (by executing one line of the code at PC_p) and moving to a new configuration, $\mathcal{C}.s$. We refer to $\mathcal{C}$ as the start configuration and $\mathcal{C}.s$ as the end configuration of the step s. A series of steps from the initial configuration, $\mathcal{C}_0$, is called a *run*. A configuration $\mathcal{C}'$ is *reachable* if the last step of some finite run of the algorithm has an end configuration of $\mathcal{C}'$. We say that a process p has *crashed* in an infinite run σ if there exists some time t such that after t, p never takes a step.

2.1 Mutual Exclusion

In concurrent programming, there are times when multiple processes wish to access a shared resource (e.g. a global file or data structure). However, accessing and modifying shared data are not always atomic actions and may take multiple steps, during which the shared resource should not be accessed by other processes since it could result in undesirable behavior or race conditions. Thus, mutual exclusion algorithms ensure that a shared resource is only available to one process at a time. The structure of mutual exclusion algorithms is commonly broken down into four sections:

- **Try Section:** the section of code that a process executes when it is attempting to access the shared resource. The Try Section is composed of two parts: a *doorway* and a *waiting room*. The *doorway* is a section of code that runs in a bounded number of steps in which a process declares its presence and wish to access the shared resource to the other processes. Immediately following the doorway, the process enters the *waiting room*, where it busy-waits until it is granted permission to enter the CS or decides to abort.

- **Critical Section (CS):** the section of code where the shared resource can be accessed.
- **Exit Section:** the section of code that a process executes just after it leaves the Critical Section, informing other processes that it has finished using the shared resource.
- **Remainder Section:** the section of code that a process executes between the Exit Section and the Try Section when the process has no interest in using the shared resource.

2.2 Reader-Writer Problem

The reader-writer problem [6] is a variation of the mutual exclusion program where the processes involved are divided into two classes: readers and writers. If a reader is in the Critical Section (CS) or is enabled to enter the CS, then other readers can be in the CS at the same time. However, if a writer is in the CS, no other readers or writers can be in the CS at the same time.

Each time a process p tries to access the shared resource is called an *attempt*. An attempt lasts from the moment p enters the Try Section to the moment p exits either the Exit Section or the Abort Section. A *read attempt* is an attempt from a reader, and a *write attempt* is an attempt from a writer.

We also introduce the terms *doorway precedes*, *doorway concurrent*, and *enabled*, which we shall use in our description of the properties for the reader-writer problem, modified from Bhatt and Jayanti's paper[2] to allow for abort:

Definition 1. If A and A' are any two attempts in a run (possibly by different processes), A *doorway precedes* A' if A completes the doorway before A' begins the Try Section. A and A' are *doorway concurrent* if neither doorway precedes the other.

Definition 2. A process p is *enabled* to enter the CS in configuration $\mathcal{C}$ if p is in the Try Section in $\mathcal{C}$ and there is an integer b such that, in all runs from $\mathcal{C}$, p enters the CS in at most b of its own steps if p does not choose to abort in any of those steps.

Effective reader-writer algorithms must satisfy a set of useful properties beyond solving the basic problem. For consistency, the following list of desirable properties in a reader-writer system is replicated almost in verbatim from Bhatt and Jayanti's paper [2], with edits made to allow for the notion of abortability:

(P1) **Mutual Exclusion:** If a writer is in the CS at any time, then no other process is in the CS at the same time.

(P2) **Bounded Exit:** There is an integer b such that in every run, every process completes the Exit section in at most b of its own steps.

(P3) **First-Come-First-Served (FCFS) among writers:** If w and w' are any two write attempts in a run and w doorway precedes w', then w' does not enter the CS before w if w does not abort during its attempt.

(P4) **First-In-First-Enabled (FIFE) among readers:** Let r and r' be any two read attempts in a run such that r doorway precedes r'. If r' enters the CS before r, then r is enabled to enter the CS at the time r' enters the CS.

(P5) **Concurrent Entering:** Informally, if all writers are in the Remainder section, readers should not experience any waiting, i.e., every reader in the Try Section should be able to proceed to the CS in a bounded number of its own steps. More precisely, there is an integer b such that, if σ is any run from a reachable configuration such that all writers are in the Remainder section in every configuration in σ, then every read attempt in σ executes at most b steps of the Try section before entering the CS.

(P6) **Livelock-freedom:** If no process crashes in an infinite run, then infinitely many attempts complete in that run.

2.3 Reader Priority

There are a number of ways to specify the problem depending on the priorities given to the two classes of processes in accessing the shared resource. The three most common specifications include reader-priority (where readers have priority over writers), writer-priority (where writers have priority over readers), and fair-switching or starvation-free (where neither class has priority over the other, and any process wishing to access the shared resource will eventually get its turn). This thesis studies only the reader-priority specification of the problem.

We begin by providing Bhatt and Jayanti's definition of $>_{rp}$ in verbatim [2]:

Definition 3. Let r and w be a read attempt and a write attempt, respectively, in a run. We define $r >_{rp} w$ if r doorway precedes w, or there is a time when some reader or writer is in the CS, r is in the waiting room, and w is in the Try Section.

We now provide Bhatt and Jayanti's definiton of reader priority in verbatim as follows [2]:

(RP1) **Reader Priority :** Let r and w be a read attempt and a write attempt, respectively, in a run. If $r >_{rp} w$, then w does not enter the CS before r.

(RP2) **Unstoppable Reader Property :** Let $\mathcal{C}$ be any reachable configuration in which some read attempt r is in the waiting room. Then we have:

1. If a reader is in the CS in $\mathcal{C}$, then r is enabled to enter the CS in $\mathcal{C}$.
2. If no writer is in the CS or the Exit section in $\mathcal{C}$ and $r >_{rp} w$ holds for all write attempts w that are in the Try Section in $\mathcal{C}$, then r is enabled to enter the CS in $\mathcal{C}$.

2.4 Abortability of Reader-Writer Algorithms

The main feature of our algorithm is the addition of the abortability properties, which allows processes in the waiting room to call the Read-Abort or Write-Abort functions, and return to the Remainder Section. We define the properties as follows:

(A1) **Wait-Free Reader-Abort:** A reader in a busy-wait loop in the Try Section can decide to execute the Read-Abort function. We require that a process be able to complete the execution of Read-Abort in a bounded number of its own steps and subsequently enter the Remainder Section.

(A2) **Wait-Free Writer-Abort:** A writer in a busy-wait loop in the Try Section can decide to execute the Write-Abort function. We require that a process be able to complete the execution of Write-Abort in a bounded number of its own steps and subsequently enter the Remainder Section.

2.5 Specification of the Abortable Reader-Priority Reader-Writer Algorithm

In this thesis, we will show algorithms that solve the reader-writer problem, fulfilling properties (P1), (P2), (P3), (P4), (P5), (P6), (RP1), (RP2), (A1), (A2) for Cache-Coherent systems with different Compare-And-Swap caching models.

2.6 Explanation of RMR (Remote Memory References) complexity

In concurrent systems, processes have access to both remote memory and local memory. While accesses to local memory are relatively fast, accesses to remote memory must travel over a network and cause delays and congestion. Thus, time complexity in concurrent systems is measured in terms of the number of Remote Memory References (RMR). In essence, local memory accesses take 0-RMRs while each access to remote memory will take 1-RMR. Thus, efficient concurrent algorithms must aim to reduce the number of RMRs made.

However, what is considered to be an RMR varies depending on the underlying memory system of the machine running the algorithm. Among modern systems, there are two prevalent shared memory models: the Distributed Shared Memory (DSM) model (also known as the Non-Uniform Memory Access (NUMA) model) and the Cache-Coherent (CC) model.

In the DSM model, each process has its own local memory and when a process accesses its local memory, it incurs 0-RMRs. In addition, a process can also access the local memory of *other* processes. However, an access to the local memory of *another* process is remote and thus incurs 1-RMR.

In the CC model, shared memory is remote to all processes, but each process is equipped with a local cache. Accesses to a process' local cache incur 0-RMR. When a process p accesses a shared variable v from *shared* memory for the first time (making 1-RMR), p will store a copy of v in its cache. As long as no processes change the value of v, future reads of v by p will be made to its cache instead of the shared memory, incurring 0-RMR. However, if another process q makes a change to v, q will incur 1-RMR for writing to the shared memory and invalidating all the caches containing a copy of v. Thus, the next time p accesses v, p incurs 1-RMR in order to transfer the new value of v into p's cache.

In previous research, Danek and Hadzilacos showed that a sublinear-RMR reader-writer al-

gorithm is not possible for the DSM model [7]. Thus, our focus for this thesis will be on the constant-RMR implementation for the CC model.

3 Hardware Support

For our algorithm, we assume that the following set of atomic operations are supported by the hardware. These atomic operations are supported in most CPUs, including x86 [1] and SPARC v9 [15] systems.

3.1 Registers

Let A be a register with a value of c. A supports the following functions:

- $Read(A)$: Return c.
- $Write(A, v)$: Set $c := v$.

If j is a local variable and K is a shared variable, we equate $j \leftarrow K$ with $j = Read(K)$. Similarly, we equate $K \leftarrow j$ with $Write(K, j)$.

3.2 Fetch and Add (F&A)

Let A be a F&A object with a value of c. A supports the following functions:

- $Read(A)$: Return c.
- $Write(A, v)$: Set $c := v$.
- $F\&A(A, v)$: Return c and set $c := c + v$.

3.3 Compare and Swap (CAS)

Let A be a F&A object with a value of c. A supports the following functions:

- $Read(A)$: Return c.
- $Write(A, v)$: Set $c := v$.
- $CAS(A, u, v)$: If $c = y$, set $c := v$ and return $true$. Otherwise, return $false$.

3.4 Cache Invalidation Models in Cache-Coherent Systems

The cache invalidation scheme in Cache-Coherent (CC) systems can vary depending on the underlying hardware system. In the case with CAS, a failed CAS by a process may or may not invalidate the caches of the other processes. This discrepancy causes different hardware systems to have different RMR complexities when running the same algorithm. To discuss this issue, we define two models describing how CAS operates in CC systems:

- **Smart-Cache Model for CAS**: In any $CAS(A, u, v)$ operation where a CAS fails (and thus not changing the value of A), the system does not invalidate any caches containing A.
- **Conservative-Cache Model for CAS**: In any $CAS(A, u, v)$ operation where a CAS fails (and thus not changing the value of A), the system may or may not invalidate any caches containing A.

Although we hope to design algorithms for the Conservative-Cache model (which has fewer hardware requirements), this prevents us from busy-waiting on any CAS variables, which is a hinderance for our algorithm.

We shall later show that our algorithm has constant-RMR for CC systems with the Smart-Cache model for CAS and amortized constant-RMR for CC systems with the Conservative-Cache model for CAS (see Section 3.4). We also present an adjustment to the algorithm for it to have constant-RMR in the Conservative-Cache model for up to R readers (where R is a constant), beyond which the RMR-complexity algorithm will be $O(\frac{n_r}{R})$, where n_r is the number of actual readers in the algorithm.

4 Previous Work

4.1 Constant RMR Reader-Priority Multi-Reader Single-Writer Algorithm

We chose Bhatt and Jayanti's constant-RMR reader-priority multi-reader single-writer algorithm [2] as the basis for our abortable algorithm because to our knowledge, it is the only constant-RMR reader-priority algorithm satisfying (P1), (P2), (P3), (P4), (P5), (P6), (RP1), (RP2). Prior to Bhatt's work, there were a number of reader-writer algorithms previously proposed for Cache-Coherent systems. However, these algorithms either do not satisfy concurrent entering [16, 11], or has linear RMR [3, 4] or $O(logn_r)$ RMR complexity (where n_r is the number of readers in the system)[8]. To the best of our knowledge, abortable reader-writer algorithms for cache-coherent systems do not yet exist.

Figure 1 on page 15 shows Bhatt and Jayanti's algorithm. The figure is attached for ease of comparing the abortable version with the unabortable version of the algorithm for readers already familiar with Bhatt and Jayanti's algorithm. For a detailed description and proof of the unabortable algorithm, refer to [2].

4.2 Abortable Mutual Exclusion Algorithm

For the multi-reader multi-writer construction of our algorithm, we can use any abortable mutual exclusion algorithm in conjunction with our single-writer multi-reader construction. The RMR-complexity of the resulting multi-reader multi-writer algorithm will depend directly on the RMR-complexity of the abortable mutual exclusion algorithm. This RMR limitation is acceptable because in the absence of readers, the abortable multi-reader multi-writer algorithm is equivalent to the abortable mutual exclusion problem.

Since Scott and Scherer proposed the need for abortability in mutual exclusion algorithms [18], several FCFS algorithms have been introduced in the literature satisfying $O(logn)$-RMR complexity, where n is the number of processes in the system [10, 14]. Any of these algorithms can be used with our construction of the multi-reader multi-writer algorithm.

As an example, we will assume that we can use the abortable mutual exclusion algorithm by Jayanti [10] to produce a $min(k, log(n_w))$-RMR algorithm for abortable multi-reader multi-writer algorithm (where k is the number of writers contending to enter the CS and n_w is the total number

Constants:

PID is the set of process IDs

Variables:

$D \in \{0,1\}$ is a read/write variable, initialized to 0
$Gate \in \{0,1\}$is a read/write variable, initialized to 0
$X \in PID \cup \{true\}$ is a CAS variable, initialized to any PID
$Permit \in \{true, false\}$ is a CAS variable, initialized to $true$
C is a fetch&add variable, initialized to 0

```
procedure Write-Lock_i(){
        REMAINDER SECTION
 1.     prevD ← D
 2.     currD ← ¬prevD
 3.     D ← currD
 4.     Permit ← false
 5.     Promote_i()
 6.     wait till Permit
        CRITICAL SECTION
 7.     Gate ← currD
 8.     X ← i
}

procedure Promote_i(){
 9      x ← X
 10.      if(x ≠ true)
 11.        if(CAS(X, x, i))
 12.          if(Permit ≠ true)
 13.            if(C = 0)
 14.              if(CAS(X, i, true))
 15.                Permit ← true
}
```

```
procedure Read-Lock_i(){
        REMAINDER SECTION
 16.    F&A(C, 1)
 17.    d ← D
 18.    x ← X
 19.    if(x ∈ PID)
 20.      CAS(X, x, i)
 21.    if(X = true)
 22.      wait till Gate = d
        CRITICAL SECTION
 23.    F&A(C, −1)
 24.    Promote_i()
}
```

Figure 1: Bhatt and Jayanti's [Unabortable] Reader-Priority Multi-Reader Single-Writer Algorithm [2]

of writers). To our knowledge, this is the most efficient abortable mutual-exclusion algorithm for the CC system and it uses LL/SC objects, which can be implemented by CAS [17].

5 Abortable Reader-Priority Single-Writer Multi-Reader Algorithm

Figure 2 on page 18 shows our abortable reader-priority single-writer multi-reader algorithm satisfying properties (P1), (P2), (P3), (P4), (P5), (P6), (RP1), (RP2), (A1), (A2).

The labels to the left of the algorithm ($w1, w2, etc.$) represent the atomic steps of the algorithm. Note that we have combined local computations into the same atomic steps as remote operations (e.g. Line $w3$) in order to shorten the length and complexity of our subsequent proof. This is permitted because local operations by a process p do not affect the global state and thus other processes cannot distinguish if the step has or has not been executed.

The algorithm in Figure 2 on page 18 shows five procedures: *Write-Lock*, *Read-Lock*, *Write-Abort*, *Read-Abort*, and *Promote*. The *Write-Lock* and *Read-Lock* procedures provides the code for the Try and Exit Sections of the writers and readers, respectively. The *Write-Abort* and *Read-Abort* procedures are called from either $w6$ or $r6$ to allow writers and readers to abort, respectively. Finally, the *Promote* procedure is used by both writers and readers to attempt to promote a writer into the Critical Section if there are no enabled readers and if the writer is waiting for permission.

5.1 Description of the Variables and Their Purpose

D The "Direction" variable used to inform readers of the side (0 or 1) on which it should enter and wait if required.

Gate The Gate controls the direction (0 or 1) in which readers are allowed to enter the CS. As observed from Line $r6$, a reader r trapped in the waiting room must wait until the Gate is equal to the D value that r previously read before r can access the CS.

X A tuple in the form $[x_1, x_2]$. We shall use x_1 and x_2 to address the individual components of X in the rest of this thesis. x_1 is used as a way to determine if the writer is interested in entering the CS, to prevent bad interleavings, and for readers to steal permission from the writers in the Try Section. When x_1 is *true*, the writer either already has permission to enter the CS or is in the Remainder Section. In both cases, the writer does not want any help in getting permission to enter the CS and readers much check the Gate to see if it is allowed to enter the CS or if it must wait for the writer. When

Constants:

PID is the set of process IDs
$nPID$ is any value such that $nPID \notin PID \cup \{true\}$

Variables:

$D \in \{0,1\}$ is a read/write variable, initialized to 0
$Gate \in \{0,1\}$is a read/write variable, initialized to 0
$X \in [(PID \cup \{true, nPID\})x(PID \cup \{nPID\})]$ is a CAS variable, initialized to $[true, nPID]$
$Permit \in PID \cup \{true, nPID\}$ is a CAS variable, initialized to $true$
C is a fetch&add variable, initialized to 0

Persistent Variables:

Each Process has a $SafeID \in PID$, initialized to i, the PID of the process.

```
procedure Write-Lock_i(){
  w_ncs: REMAINDER SECTION
  w1:    [*, b] ← X
  w2:    X ← [i, b]
  w3:    prevD ← D
         currD ← ¬prevD
  w4:    D ← currD
  w5:    Permit ← b
  wp:    Promote_i()
  w6:    wait till Permit
                   or Write-Abort_i()
  w_cs:  CRITICAL SECTION
  w7:    Gate ← currD
}

procedure Read-Lock_i(){
  r_ncs: REMAINDER SECTION
  r1:    F&A(C, 1)
  r2:    d ← D
  r3:    [a, b] ← X
  r4:    if(a ∈ PID)
           CAS(X, [a, b], [i, b])
  r5:    if(X = [true, *])
  r6:      wait till Gate = d
                   or Read-Abort_i()
  r_cs:  CRITICAL SECTION
  r7:    F&A(C, −1)
  rp:    Promote_i()
}

procedure Write-Abort_i(){
  wa1:   Permit ← true
  wa2:   [a, b] ← X
  wa3:   if(a ≠ true)
           CAS(X, [a, b], [nPID, b])
  wa4:     if(X ≠ [true, *])
  wa5:       D ← prevD
             return
  wa6:   Gate ← currD
}

procedure Read-Abort_i(){
  ra1:   F&A(C, -1)
  rap:   Promote_i()
}

procedure Promote_i(){
  f1:    [a, b] ← X
         if(a ≠ true)
  f2:      if(CAS(X, [a, b], [i, b]))
  f3:        if(Permit ≠ true)
  f4:          if(C = 0)
  f5:            if(CAS(X, [i, b], [true, SafeID]))
                   SafeID ← b
  f6:              CAS(Permit, b, true)
}
```

Figure 2: Abortable Reader-Priority Single-Writer Multi-Reader Algorithm. Code for process with *pid* i. The doorway for the reader is Lines $r1$ to $r5$.

x_1 is $\neg true$, it is either a *pid* or $nPID$, indicating that it *might* be attempting to enter the CS. Thus, exiting readers will attempt promote the writer into the CS under the correct conditions. On the other hand, x_2 is used to store either a value that isn't being used as a $SafeID$ by any process. This is used for ensuring that the permission given by a reader is current, and not for a previous iteration of *Write-Lock*.

Permit The permission used to determine if a writer is allowed to enter the CS or if it wants to access the CS at all. The *Permit* is what the writer busy-waits on to see if it is allowed to enter the CS. (Note that writers cannot busy-wait on X because X can change an unbounded number of times during a single iteration of $Write-Lock$). *Permit* is *true* when the writer has permission to enter the CS or when the writer does not want to access the CS at all (i.e. in the Remainder, Exit, or Abort Sections). In both cases, the writer is indicating that it does not need to be promoted into the CS. However, when the writer wishes to access the CS (i.e. in the Try or Critical Section), it sets *Permit* to the value present in the current x_2 value, which helps distinguish the iteration of $Write-Lock$ calls, to prevent readers at Line $f6$ from accidentally granting permission to the writer after the writer has aborted.

C A count of the readers in the Try Section or CS of the algorithm. This helps writers in the Try Section or leaving readers determine if there are any remaining readers. If not, the process should proceed with attempting to grant permission to the writer.

$SafeID$ A unique persistent variable (a variable that a process retains even when it completes the *Write-Lock* or *Read-Lock* procedures, so it can be used in the next call of *Write-Lock* or *Read-Lock*) stored by all processes indicating the next number that is safe to be used for x_2. This is used to prevent readers from incorrectly giving a writer permission to enter the CS if a writer has aborted and subsequently returns to the Try Section. Each time a process p executes Line $f5$ of the algorithm, it atomically switches $SafeID$ and x_2, since x_2 is no longer a safe value to be used until process p executes Line $f6$, where it tries to grant permission to the writer.

5.2 Reader's Protocol: Line-by-Line Commentary

We walk through possible steps of a reader r:

$r1$ When r enters the Try Section, it first declares its presence by incrementing C. This prevents processes at $f4$ from granting permission to the writer.

$r2$ r reads D to find out the "direction" that it belongs to. This will allow r to determine whether it has permission to enter the CS at $r6$ in the case that $X = true$.

$r3$ r reads X for the next step.

$r4$ r attempts to CAS X in order to steal permission away from any potential processes trying to promote the writer (since this is a reader-priority algorithm).

$r5$ r checks the current value of X. If it is $\neg true$, r knows it is permitted to enter the CS. If it is $true$, then the writer might have permission to enter the CS, so r must then check if the $Gate$ to its direction is open in $r6$.

$r6$ r sees if the $Gate$ for its direction is open (i.e. $Gate = D$). If it is, r can go into the CS. If it isn't, r must wait until the writer flips the $Gate$ at $w7$ or $wa6$.

$r7$ After the CS, r erases its existence by decrementing C and then running $Promote$ to see if it can give permission to the writer.

$ra1$ To abort, r simply executes the Exit Section. Identical to $r7$.

5.3 Writer's Protocol: Line-by-Line Commentary

Let the writer be denoted as w. Here is a list of the potential steps of the writer:

$w1$ w reads X for the next step.

$w2$ w sets X to w's pid, so that in the case that X was $true$, it is now a non-$true$ value, informing readers that X is interested in accessing the CS. Note, however, that since $Permit = true$ at this point and w just set x_1 to its own PID, other readers will not be able to give permission to w yet.

$w3$ w reads D in preparation for toggling it.

$w4$ w toggles D so that $D \neq Gate$. This ensures that readers who come in after w is granted permission for the CS will be blocked at $r6$. Note that if there are still readers r for which $r.d = Gate$, the writer will need to wait for all of them to leave the CS before w can enter the CS. Thus, when w is in the CS, all the readers must have the same direction.

$w5$ The $Permit$ is set to x_2. Since x_2 is only changed at Line $f5$ (where w is granted permission), and since the line will not be successfully executed until $Permit$ is set to a non-true value, we know that $w.b = x_2$ at the time of execution of $w5$.

wp w tries to give itself permission to enter the CS

$w6$ Since X can undergo many changes from $true$ to various pids during each call of $Write\text{-}Lock$, w cannot busy-wait on X because that will produce linear-RMR. Thus, w waits on $Permit$ instead. At Line $w6$, w continuously checks to see if $Permit = true$ so it can enter the CS. Notice how readers can only set $Permit$ to $true$ and nothing else, so in the Smart-Cache model, w will only have at most two cache misses at Line $w6$

$w7$ Upon exiting the CS, w can simply toggle the $Gate$ to let any waiting readers enter. Note: this will set $Gate = D$, so new readers will no longer be blocked.

$wa1$ w can choose to abort at Line $w6$. In the Abort Section, w first sets $Permit = true$ to stop readers before line $f3$ from granting w permission to enter the CS.

$wa2$ w reads X in preparation of the next step

$wa3$ w checks if x_1 is $true$. If so, w must have been granted permission to enter the CS already, so it can just execute the Exit Section at $wa6$. However, if it doesn't have permission yet, w tries to CAS $nPID$ into x_1. This is done to prevent other readers from granting w permission. If a reader is at Line $f4$ or $f5$, it must have read $Permit = true$ before w executed $wa1$. Thus, if w CASes $nPID$ into x_1 before any readers execute $f5$, it guarantees that either w or another reader before $f3$ has changed the value of x_1. Thus, Line $f5$ should fail from now on.

$wa4$ w checks to see the current value of x_1. If x_1 is *true*, a reader must have executed $f5$ before w executed Line $wa3$. Thus, w has permission to enter the CS, so it can simply execute the Exit Section at $wa6$. Otherwise, if x_1 is $\neg true$, we know that from now on, processes at $f5$ will fail. Thus, w will not be granted permission by any other processes from now on.

$wa5$ Since it is guaranteed that w will not be given permission by any readers from this point onwards, w simply reverses D. Notice that until the next iteration of *Write-Lock*, x_1 will stay $\neg true$, so readers will be able to enter the CS via $r5$.

$w6$ Identical to $w7$

5.4 Promote Procedure: Line-by-Line Commentary

The *Promote* procedure is executed by either a writer or an exiting reader in order to check if permission needs to be given to a writer. Let p be the process executing *Promote*:

$f1$ p reads X and checks if x_1 is *true*. If it is *true*, the writer has no intention of accessing the CS, so p can just leave.

$f2$ If X isn't *true*, p attempts to CAS its own *pid* into x_1 to let other processes know that it is trying to give permission to the writer. This prevents multiple processes from granting the writer permission at the same time and also allows new readers at Line $r4$ to alert p not to give the writer permission. If the CAS fails, another process must have successfully CASed X for one of the above reasons, so p can simply leave. Otherwise, p proceeds to Line $f3$.

$f3$ p checks if *Permit* is *true*. If *Permit* is *true*, the writer either does not want permission or has already been granted permission, so p can simply leave. Otherwise, p proceeds to Line $f4$.

$f4$ p checks to C to see if there are other readers between lines $r2...r7$. If so, p can leave because one of those readers will eventually grant the writer permission if necessary. Otherwise, if $C = 0$, p must be the writer or one of the last exitting readers. So it can proceed.

$f5$ p attempts to grant the writer permission by CASing x_1 with $true$ and x_2 with $p.SafeID$. If the CAS fails, then x_1 is either $true$ (in which case the writer has permission already or is not interested in the CS) or $\neg true$, in which case the process with $pid = x_1$ is either in an earlier step of $Promote$ (in which case it can grants the permission to the writer) or a reader is in the Try Section, in which case the writer should not yet be granted permission. Otherwise, if the CAS succeeds, the writer now has permission to enter the CS. Note that p also switches the values of x_2 and $p.SafeID$. This provides the writer with a safe x_2 value to use as a $Permit$ in the future. Also, since the CAS at Line $f5$ can only be successfully executed once for every call of $Write\text{-}Lock$, only one process can swap b with its own $SafeID$, and only one process will have that b value at Line $f6$.

$f6$ p tries to grant permission to the waiting writer. $f6$ should succeed unless if the writer decided to abort from $Write\text{-}Abort$ during that iteration, in which case p will fail at Line $f6$ because $p.b$ will either be set to true or another process' $SafeID$, which must be distinct from $p.b$, which is equal to $p.SafeID$.

6 Model Checking

In the prototyping stage of algorithm design, we used a series of specification and modelling tools (PlusCal, TLA+, and TLC) to verify the correctness of our algorithm.

Appendix Section A shows the PlusCal, TLA+ specification of our abortable reader-priority single-writer multi-reader algorithm and its invariants.

6.1 PlusCal

PlusCal[13] (formerly +Cal) is a language that allows programmers to formally specify both concurrent and sequential algorithms and translate it into TLA+ (6.2). PlusCal allows programmers to specify their algorithms at a high level, providing a very flexible granularity of atomicity (for concurrent systems).

We first designed our algorithm in PlusCal and used the built-in translator to translate our algorithm to TLA+. Since PlusCal is a relatively high-level language built with concurrency in mind, our experience with PlusCal was very positive and it was easy to make changes confidently in PlusCal without having to change much of the code.

6.2 TLA+

TLA+ (Temporal Logic of Actions) [12] is a formal specification language that defines an algorithm in terms of the initial state, subsequent states and their transitions. The advantages of using TLA+ is that languages specified in TLA+ can quickly be debugged or verified using the TLC model checker (6.3).

For this thesis, now only did we translate our algorithm from PlusCal to TLA+, we also specified our invariants (see Section 7.2) in TLA+.

6.3 TLC

TLC is a model-checking program that runs simulations of TLA+ specifications for every possible execution up to a certain collision probability and verifies any assumptions and invariants provided at each state of the execution. If an assumption or invariant is violated, TLC prints out the execution where the violation occurred. TLC can verify the correctness of a two-process mutex algorithm to a

high degree of accuracy within about 2 or 3 minutes. However, the tool has several limitations. First, it is only able to verify safety properties and not liveness properties. However, since our invariants are all safety properties, this was not an issue in our case. Second, it can only be run realistically on small datasets. For example, verifying the correctness of a single-writer 3-reader version of our algorithm took over 1 day to complete. Finally, since there are infinitely many states in the system, TLC will only model-check until the state collision rate becomes sufficient. Moreover, it can only verify executions for a given number of processes, so it merely serves as a tool for prototyping and debugging.

In the proof process, TLC was extremely useful to quickly verify new invariants or changes in invariants. In the case that an invariant is violated in some run, TLC outputs the list of configurations in the run, which was helpful to understanding where we went wrong. Furthermore, we used TLC to verify if certain lines in the algorithm could be omitted or changed. While the model checker could potentially run for a very long time, from our experience, most errors were identified within the first 10 minutes of execution.

7 Proof of Algorithm

We prove our algorithm correct by invariants. We do this in two parts. First, we will prove that the set of invariants listed in Section 7.2 is true in every reachable configuration of the algorithm. Then, we will consider the invariants as facts and use them to prove the properties of the algorithm.

7.1 Notation

Here are the notations used in defining our invariants (derived from [2]):

- We denote the value of a local variable y of process p by $p.y$. Similarly, we denote the PID of process p by $p.i$
- We let $X = [x_1, x_2]$ for the global variable X so we can address x_1 and x_2 separately in our invariants
- PC_p denotes the program counter of a process p. As there is only one writer at any time, we denote it by w and its program counter by PC_w. Note that at any time t, $PC_w \in \{w1...w7, wa1...wa6, f1...f6\}$ and for a reader r, $PC_r \in \{r1...r7, ra1, f1...f6\}$
- $R(condition\ A)$ denotes the set of readers which satisfy the *condition* A, e.g., $R(PC \in \{r3...r6\}, d \neq Gate)$ is the set of all readers r, such that $PC_r \in \{r3...r6\}$ and $r.d \neq Gate$
- $P(H)$, denotes the set of *pid*s corresponding to the processes with their PCs coming from set H. More formally, $P(H) = \{p.i : PC_p \in H\}$

7.2 Invariants

Let $I = I_G \wedge I_{w1,w2} \wedge I_{w3,w4} \wedge I_{w5} \wedge I_{f1} \wedge I_{f2} \wedge I_{f3,f4} \wedge I_{f5} \wedge I_{f6} \wedge I_{w6,wa1} \wedge I_{w7,wa6} \wedge I_{wa2} \wedge I_{wa3} \wedge I_{wa4} \wedge I_{wa5}$, where I_G is the global invariant which holds in every reachable configuration of the algorithm. The other invariants are denoted by I_A , meaning that the invariant applies when $PC_w \in \{A\}$.

Bold font is used to represent lines where the invariant has changed from step to step for ease of understanding the proof.

I_G:

1. $C = |R(PC \in \{r2...r7, ra1\})|$
2. $|\{x_2\} \cup \{SafeID_i\}| = n + 1$

$I_{w1,w2}$: $PC_w \in \{w1, w2\} \implies$

1. $Gate = D$
2. $Permit = true$
3. $x_1 = true \implies R(PC \in \{r3...r6\} \wedge d \neq Gate) = \phi$
4. $x_1 \neq true \implies R(PC = r6 \wedge d \neq Gate) = \phi$
5. $R(PC \in \{f4, f5\} \wedge x_1 = i) = \phi$

$I_{w3,w4}$: $PC_w \in \{w3, w4\} \implies$

1. $Gate = D$
2. $Permit = true$
3. $\boldsymbol{x_1 \in PID}$
4. $\boldsymbol{R(PC = r6 \wedge d \neq Gate) = \phi}$
5. $R(PC \in \{f4, f5\} \wedge x_1 = i) = \phi$

I_{w5}: $PC_w = w5 \implies$

1. $\boldsymbol{Gate = \overline{D}}$
2. $Permit = true$
3. $x_1 \in PID$
4. $R(PC = r6 \wedge d \neq Gate) = \phi$
5. $R(PC \in \{f4, f5\} \wedge x_1 = i) = \phi$

I_{f1}: $PC_w = f1 \implies$

1. $Gate = \overline{D}$
2. $\boldsymbol{Permit \in \{true, nPID\} \cup PID}$
3. $\boldsymbol{x_1 \in (PID \cup \{true\})}$
4. $\boldsymbol{R(PC = f5 \wedge x_1 = i) \neq \phi \implies}$
   (a) $\boldsymbol{R(PC \in \{r6, r7\}) = \phi}$
   (b) $\boldsymbol{R(PC \in \{r3...r5\} \wedge d = Gate) = \phi}$
5. $\boldsymbol{x_1 \in PID \implies}$
   (a) $\boldsymbol{R(PC = r6 \wedge d \neq Gate) = \phi}$
   (b) $\boldsymbol{Permit = w.b}$
6. $\boldsymbol{x_1 = true \implies}$

(a) $\boldsymbol{R(PC \in \{r3...r6\} \wedge d = Gate) = \phi}$

(b) $\boldsymbol{(Permit \neq true \implies |R(PC = f6 \wedge b = Permit)| = 1)}$

(c) $\boldsymbol{R(PC = r7) = \phi}$

I_{f2}: $PC_w = f2 \implies$

1. $Gate = \overline{D}$
2. $Permit \in \{true, nPID\} \cup PID$
3. $x_1 \in (PID \cup \{true\})$
4. $R(PC = f5 \wedge x_1 = i) \neq \phi \implies$
   (a) $R(PC \in \{r6, r7\}) = \phi$
   (b) $R(PC \in \{r3..r5\} \wedge d = Gate) = \phi$
5. $x_1 \in PID \implies$
   (a) $R(PC = r6 \wedge d \neq Gate) = \phi$
   (b) $Permit = w.b$
   (c) $\boldsymbol{x_1 \neq w.a \implies R(PC \in \{r2...r7, ra1, f1\}) \cup R(a = x_1, PC = f2) \neq \phi \vee x_1 \in P(f3...f5)}$
6. $x_1 = true \implies$
   (a) $R(PC \in \{r3..r6\} \wedge d = Gate) = \phi$
   (b) $(Permit \neq true \implies |R(PC = f6 \wedge b = Permit)| = 1)$
   (c) $R(PC = r7) = \phi$

$I_{f3,f4}$: $PC_w \in \{f3, f4\} \implies$

1. $Gate = \overline{D}$
2. $Permit \in \{true, nPID\} \cup PID$
3. $x_1 \in (PID \cup \{true\})$
4. $R(PC = f5 \wedge x_1 = i) \neq \phi \implies$
   (a) $R(PC \in \{r6, r7\}) = \phi$
   (b) $R(PC \in \{r3...r5\} \wedge d = Gate) = \phi$
5. $x_1 \in PID \implies$
   (a) $R(PC = r6 \wedge d \neq Gate) = \phi$
   (b) $Permit = w.b$
   (c) $\boldsymbol{x_1 \neq w.i \implies R(PC \in \{r2...r7, ra1, f1\}) \cup R(a = x_1, PC = f2) \neq \phi \vee x_1 \in P(f3...f5)}$
6. $x_1 = true \implies$
   (a) $R(PC \in \{r3...r6\} \wedge d = Gate) = \phi$
   (b) $(Permit \neq true \implies |R(PC = f6 \wedge b = Permit)| = 1)$
   (c) $R(PC = r7) = \phi$

I_{f5}: $PC_w = f5 \implies$

1. $Gate = \overline{D}$
2. $Permit \in \{true, nPID\} \cup PID$
3. $x_1 \in PID \cup \{true\}$
4. $\boldsymbol{x_1 \in PID \implies}$

(a) $\boldsymbol{R(PC = r6) = \phi}$

(b) $Permit = w.b$

(c) $x_1 \neq w.i \implies R(PC \in \{r2...r7, ra1, f1\} \cup R(a = x_1, PC = f2) \neq \phi \vee x_1 \in P(f3...f5)$

5. $\boldsymbol{R(PC \in \{r3...r6\} \wedge d = Gate) = \phi}$

6. $x_1 = true \implies$

(a) $(Permit \neq true \implies |R(PC = f6 \wedge b = Permit)| = 1)$

7. $R(PC = r7) = \phi$

I_{f6}: $PC_w = f6 \implies$

1. $Gate = \overline{D}$

2. $\boldsymbol{Permit = w.b}$

3. $\boldsymbol{x_1 = true}$

4. $R(PC \in \{r3...r6\} \wedge d = Gate) = \phi$

5. $\boldsymbol{R(PC = f6 \wedge b = Permit) = \phi}$

6. $R(PC = r7) = \phi$

$I_{w6,wa1}$: $PC_w \in \{w6, wa1\} \implies$

1. $Gate = \overline{D}$

2. $Permit \in \{true, nPID\} \cup PID$

3. $x_1 \in (PID \cup \{true\})$

4. $R(PC = f5 \wedge x_1 = i) \neq \phi \implies$

(a) $R(PC \in \{r6, r7\}) = \phi$

(b) $R(PC \in \{r3...r5\} \wedge d = Gate) = \phi$

5. $x_1 \in PID \implies$

(a) $R(PC = r6 \wedge d \neq Gate) = \phi$

(b) $Permit = w.b$

(c) $\boldsymbol{R(PC \in \{r2...r7, ra1, f1\} \cup R(a = x_1, PC = f2) \neq \phi \vee x_1 \in P(f3...f5)}$

6. $x_1 = true \implies$

(a) $R(PC \in \{r3...r6\} \wedge d = Gate) = \phi$

(b) $(Permit \neq true \implies |R(PC = f6 \wedge b = Permit)| = 1)$

(c) $R(PC = r7) = \phi$

$I_{w7,wa6}$: $PC_w \in \{w7, wa6\} \implies$

1. $Gate = \overline{D}$

2. $\boldsymbol{Permit = true}$

3. $x_1 = true$

4. $R(PC \in \{r3...r6\} \wedge d = Gate) = \phi$

5. $R(PC = r7) = \phi$

I_{wa2}: $PC_w = wa2 \implies$

1. $Gate = \overline{D}$

2. $\boldsymbol{Permit = true}$

3. $x_1 \in (PID \cup \{true\})$

4. $R(PC = f5 \wedge x_1 = i) \neq \phi \implies$

(a) $R(PC \in \{r6, r7\}) = \phi$

(b) $R(PC \in \{r3...r5\} \wedge d = Gate) = \phi$

5. $\boldsymbol{x_1 \in PID \implies}$

   (a) $\boldsymbol{R(PC = r6 \wedge d \neq Gate) = \phi}$

6. $x_1 = true \implies$

   (a) $R(PC \in \{r3...r6\} \wedge d = Gate) = \phi$

   (b) $R(PC = r7) = \phi$

I_{wa3}: $PC_w = wa3 \implies$

1. $Gate = \overline{D}$
2. $Permit = true$
3. $x_1 \in (PID \cup \{true\})$
4. $R(PC = f5 \wedge x_1 = i) \neq \phi \implies$

   (a) $R(PC \in \{r6, r7\}) = \phi$

   (b) $R(PC \in \{r3...r5\} \wedge d = Gate) = \phi$

5. $x_1 \in PID \implies$

   (a) $R(PC = r6 \wedge d \neq Gate) = \phi$

   (b) $\boldsymbol{x_1 \neq w.a \implies R(PC \in \{f4, f5\} \wedge x_1 = i) = \phi}$

6. $x_1 = true \implies$

   (a) $R(PC \in \{r3...r6\} \wedge d = Gate) = \phi$

   (b) $R(PC = r7) = \phi$

I_{wa4}: $PC_w = wa4 \implies$

1. $Gate = \overline{D}$
2. $Permit = true$
3. $\boldsymbol{x_1 \in (PID \cup \{true, nPID\})}$
4. $\boldsymbol{R(PC \in \{f4, f5\} \wedge x_1 = i) = \phi}$
5. $x_1 \neq true \implies R(PC = r6 \wedge d \neq Gate) = \phi$
6. $x_1 = true \implies$

   (a) $R(PC \in \{r3...r6\} \wedge d = Gate) = \phi$

   (b) $R(PC = r7) = \phi$

I_{wa5}: $PC_w = wa5 \implies$

1. $Gate = \overline{D}$
2. $Permit = true$
3. $\boldsymbol{x_1 \neq true}$
4. $R(PC \in \{f4, f5\} \wedge x_1 = i) = \phi$
5. $\boldsymbol{R(PC = r6 \wedge d \neq Gate) = \phi}$

7.3 Proof of Invariants

To prove the invariants correct, we must prove that:

- I holds in the initial configuration $\mathcal{C}_0$, and
- if I holds in a reachable configuration $\mathcal{C}$, and if some process takes a step s, I also holds in $\mathcal{C}.s$.

Lemma 4. *I holds in $\mathcal{C}_0$.*

Proof. Let $ReaderSet$ be the set of all readers and let w represent the writer. It suffices to prove that I_G and I_{w1w2} hold in $\mathcal{C}_0$, since all other invariants hold trivially by the fact that in $\mathcal{C}_0$, $PC_w = w1$ and $\forall r \in ReaderSet : PC_r = r1$. □

Claim 5. I_G holds in $\mathcal{C}_0$:

Proof. Item 1 of I_G holds because $C = 0$ initially and we observe that $|R(PC \in \{r2...r7, ra\})| = 0$. Item 2 of I_G holds since $x_2 = nPID$ and $\forall p \in ReaderSet \cup \{w\}, p.SafeID = p.i$, and $nPID \notin PID$ by definition. □

Claim 6. $I_{w1,w2}$ holds in $\mathcal{C}_0$:

Proof. Items 1,2 of $I_{w1,w2}$ hold by the initial specification of $\mathcal{C}_0$. Item 3 holds because $\forall r \in ReaderSet : PC_r = r1$. Items 4,5 hold trivially since $x_1 = true$ in $\mathcal{C}_0$. □

Lemma 7. *If I holds in $\mathcal{C}$, then I holds in $\mathcal{C}.s$, where $\mathcal{C}.s$ is the configuration after some process p took a step s in $\mathcal{C}$.*

Claim 8. If I holds in $\mathcal{C}$, I_G holds in $\mathcal{C}.s$

Proof. Item 1: by simple observation of the algorithm, we see that $\mathcal{C}$ is only incremented at Line $r1$ and decremented at Line $r7$ or $ra1$. Thus, $C = |R(PC \in \{r2...r7, ra1\})|$. Item 2: By observation, we only switch $SafeID_i$ and x_2 at Line $f5$ atomically, so if Item 2 of I_G holds in $\mathcal{C}$, it will continue to hold in $\mathcal{C}.s$. □

For the rest of the invariants, we will use the following table of proofs for clarity.

In an entry where Invariant = A, w = B, r = D, Step = E, and Item = F, the context of the proof presented is as follows: If I holds in configuration $\mathcal{C}$, then Item F of A holds in $\mathcal{C}.s$, where $PC_w \in B$, $PC_r \in D$, and E takes a step.

For example, in entry 3 of the invariant table below, the context of the proof is as follows: If I holds in $\mathcal{C}$, then Item 3 of $I_{w1,w2}$ holds in $\mathcal{C}.s$ when $PC_w \in \{w1, w2\}$, $PC_r \in \{r2\}$, and r takes a step.

Invariant	w	r	Step	Item	Proof
$I_{w1,w2}$	$w1$	-	w	all	Since $I_{w1,w2}$ holds in $\mathcal{C}$, it continues to hold in $\mathcal{C}.s$ after step $w1$. From here onwards, we will simply say this is obvious.
$I_{w1,w2}$	$w1$, $w2$	$r1, r3, r6$, $r7, ra1, f1$, $f4, f6$	r	all	obvious
$I_{w1,w2}$	$w1$, $w2$	$r2$	r	3	In $\mathcal{C}$, Gate = D by $I_{w1,w2}$, and d=D by step $r2$, so d=Gate in $\mathcal{C}.s$, as req'd
$I_{w1,w2}$	$w1$, $w2$	$r2$	r	rest	obvious
$I_{w1,w2}$	$w1$, $w2$	$r4$	r	3,4	If $x_1 = true$ in $\mathcal{C}$, the global vars remain unchanged in $\mathcal{C}.s$, so Items 3, 4 will hold. If $x_1 \neq true$ in $\mathcal{C}$, the CAS in step $r4$ will succeed but still, $x_1 \neq true$ in $\mathcal{C}.s$, so Items 3, 4 will hold.
$I_{w1,w2}$	$w1$, $w2$	$r4$	r	rest	obvious
$I_{w1,w2}$	$w1$, $w2$	$r5$	r	3,4	If $x_1 = true$ in $\mathcal{C}$, Item 3 in $\mathcal{C}.s$ follows from Item 3 of $I_{w1,w2}$ in $\mathcal{C}$ and Item 4 holds trivially. If $x_1 \neq true$ in $\mathcal{C}$, by step $r5$, r will not enter Line $r6$, trivially satisfying Items 3, 4.

$I_{w1,w2}$	$w1$, $w2$	$r5$	r	rest	obvious
$I_{w1,w2}$	$w1$, $w2$	$r5$	r	3,4	Proof analogous to that of $I_{w1,w2}$, step $r4$, Items 3, 4
$I_{w1,w2}$	$w1$, $w2$	$r5$	r	rest	obvious
$I_{w1,w2}$	$w1$, $w2$	$f3$	r	5	In $\mathcal{C}$, since $Permit = true$ by Item 2 of $I_{w1,w2}$, r will not step into Line $f4$, satisfying Item 5
$I_{w1,w2}$	$w1$, $w2$	$f3$	r	rest	obvious
$I_{w1,w2}$	$w1$, $w2$	$f5$	r	3,4	By Item 5 of $I_{w1,w2}$ in $\mathcal{C}$, the CAS in step $f5$ must fail. Thus, r will not step into Line $f6$.
$I_{w1,w2}$	$w1$, $w2$	$f5$	r	rest	obvious
$I_{w3,w4}$	$w2$	-	w	3	Follows from step $w2$
$I_{w3,w4}$	$w2$	-	w	4	Follows from Item 4 of $I_{w1,w2}$ and step $w2$
$I_{w3,w4}$	$w2$	-	w	rest	obvious
$I_{w3,w4}$	$w3$	-	w	all	obvious
$I_{w3,w4}$	$w3$, $w4$	all	r	all	Proof analogous to that of $I_{w1,w2}$
I_{w5}	$w4$	-	w	1	Follows from Item 1 of $I_{w3,w4}$ and step $w4$
I_{w5}	$w4$	-	w	rest	obvious
I_{w5}	$w5$	all	r		Proof analogous to that of $I_{w1,w2}$
I_{f1}	$w5$	-	w	2	Follows from step $w5$ and the fact that $x_2 \in \{nPID\} \cup PIDs$.
I_{f1}	$w5$	-	w	4	Follows trivially from Item 5 in I_{w5}
I_{f1}	$w5$	-	w	5a	Follows from Item 4 of I_{w5}
I_{f1}	$w5$	-	w	5b	Follows from step $w5$

I_{f1}	$w5$	-	w	6	Trivially holds by Item 3 of I_{w5}
I_{f1}	$w5$	-	w	rest	obvious
I_{f1}	$f1$	$r1, r3, r7,$ $ra1, f1, f3$	r	all	obvious
I_{f1}	$f1$	$r2$	r	4b, 6a	$d = D$ by step $r2$, and $Gate = \overline{D}$ in $\mathcal{C}$ by I_{f1}, so $d \neq Gate$ in $\mathcal{C}.s$
I_{f1}	$f1$	$r2$	r	rest	obvious
I_{f1}	$f1$	$r4$	r	all	Proof analogous to that of $I_{w1,w2}$, step $r4$
I_{f1}	$f1$	$r5$	r	4a	We will show that if there is a reader at $r5$, $R(PC = f5 \wedge x_1 = i) = \phi$, so Item 4a will trivially hold. Assume the contradiction and let r_{f5} be the reader for which $PC_{r_{f5}} = f5$ and $x_1 = r_{f5}.i$ and $PC_r = r5$ at time t. We observe that r_{f5} must have executed Line $f4$ before r executed Line $r1$ because r_{f5} read $\mathcal{C}$ to be 0. Thus, the value of x_1 that r read at Line $r3$ must be $r_{f5}.i$ or another *pid* that overwrote $r_{f5}.i$. At Line $r4$, either r successfully CASes and overwrites the value of x_1, or another process wrote a different value into x_1. From inspection of the code, since no other processes would write $r_{f5}.i$ into x_1 besides r_{f5} itself, the value of $x_1 \neq r_{f5}.i$ at time t, which is a contradiction.
I_{f1}	$f1$	$r5$	r	5a	If $x_1 = true$, Item 5a trivially holds If $x_1 \neq true$, by step $r5$, r will not step into $r6$
I_{f1}	$f1$	$r5$	r	6a	Follows from Item 6a of I_{f1} in $\mathcal{C}$
I_{f1}	$f1$	$r5$	r	6c	If $x_1 \neq true$, Item 6c trivially holds If $x_1 = true$, by step $r5$, r will not step into $r7$

I_{f1}	$f1$	$r5$	r	rest	obvious
I_{f1}	$f1$	$r6$	r	4a	Follows from Item 4a of I_{f1} in $\mathcal{C}$
I_{f1}	$f1$	$r6$	r	6c	Follows from Item 6a of I_{f1} in $\mathcal{C}$
I_{f1}	$f1$	$r6$	r	rest	obvious
I_{f1}	$f1$	$f2$	r	all	Proof analogous to that of $I_{w1,w2}$, step $r4$
I_{f1}	$f1$	$f4$	r	4	Follows from Item 1 of I_G, and step $f4$
I_{f1}	$f1$	$f4$	r	rest	obvious
I_{f1}	$f1$	$f5$	r	6a	Follows from Item 4b of I_{f1}
I_{f1}	$f1$	$f5$	r	6b	Assume by contradiction that in $\mathcal{C}.s$, $Permit \neq true \implies \lvert R(PC = f6 \wedge b = Permit)\rvert = 2$. Since $SafeID = b$ by step $f5$, the two processes in $R(PC = f6 \wedge b = Permit)$ must have the same $SafeID$, which contradicts Item 2 of I_G
I_{f1}	$f1$	$f5$	r	6c	Follows from Item 4a of I_{f1}
I_{f1}	$f1$	$f5$	r	rest	obvious

I_{f1}	$f1$	$f6$	r	5b	We will show that the CAS at Line $f6$ will only succeed if $x_1 = true$. Assume by contradiction that r successfully CASes $Permit$ at Line $f6$ at time t and $x_1 \neq true$. We know that r had successfully set x_1 to $true$ in step $f5$. By observation of the code, only the writer can change x_1 from $true$ to a non-$true$ value. Thus, the writer must have executed lines $w2$ to just before $f1$ after r executed Line $f5$. Thus, it must have executed Line $w5$, setting $Permit$ to x_2. However, since $b = r.SafeID$ by step $f5$ of r, by Item 2 of I_G, $x_2 \neq r.SafeID$, so $x_2 \neq r.b$ while r is at Line $f6$. Thus, the CAS at Line $f6$ fails, which is a contradiction.
I_{f2}	$f1$	-	w	5c	Trivially true since $w.a = x_1$ by step $f1$
I_{f2}	$f1$	-	w	rest	obvious
I_{f2}	$f2$	$r1...r3$, $r5...r7, ra1$, $f6$	r	all	Analogous to proof for I_{f1} Note: Item 5c is satisfied for these steps by Item 5c of I_{f2} being satisfied in $\mathcal{C}$
I_{f2}	$f2$	$r4$	r	5c	r satisfies Item 5c
I_{f2}	$f2$	$r4$	r	rest	Analogous to proof for I_{f1}
I_{f2}	$f2$	$f1$	r	5c	If $r.a \neq true$ in $\mathcal{C}$, then r will enter the set $R(a = x_1, PC = f2)$ by step $f1$, satisifying Item 5c. If $r.a = true$ in $\mathcal{C}$, since $x_1 = r.a$ (apparent in step $f1$), $x_1 = true$, so Item 5 is satisfied trivially.
I_{f2}	$f2$	$f1$	r	rest	Analogous to proof for I_{f1}

I_{f2}	$f2$	$f2$	r	5c	If $x_1 = r.a$ in $\mathcal{C}$, then it will satisfy $x_1 \in P(f3...f5)$ in $\mathcal{C}.s$ by Item 5c of I_{f2} and step $f2$. If $x_1 \neq r.a$ in $\mathcal{C}$, by inspection of Item 5c of I_{f2} and step $f2$, r must not b the reader satisfying Item 5c in $\mathcal{C}$. Thus, there must exist another reader r' satisfies Item 5c in $\mathcal{C}$ and thus, r' will still satisfy Item 5c in $\mathcal{C}.s$.
I_{f2}	$f2$	$f2$	r	rest	Analogous to proof for I_{f1}
I_{f2}	$f2$	$f3$	r	5c	If $x \notin PID$ in $\mathcal{C}$, Item 5 is satisfied trivially. If $x \in PID$ in $\mathcal{C}$, by Item 5b I_{f2}, $Permit = w.b$, so step $f3$ must succeed, satisfying Item 5c.
I_{f2}	$f2$	$f3$	r	rest	Analogous to proof for I_{f1}
I_{f2}	$f2$	$f4$	r	5c	If $C = 0$ in $\mathcal{C}$, step $f4$ succeeds, satisfying 5c. If $C \neq 0$ in $\mathcal{C}$, by I_G, $\lvert R(PC \in \{r2...r7, ra1\})\rvert \neq 0$. Since the cardinality of the set is always positive, some process in $R(PC \in \{r2...r7, ra1\}$ satisfies I_{f2} in $\mathcal{C}$ and will satisfy I_{f2} in $\mathcal{C}.s$ also.
I_{f2}	$f2$	$f4$	r	rest	Analogous to proof for I_{f1}
I_{f2}	$f2$	$f5$	r	5	If $x_1 = r.i$ in $\mathcal{C}$, $x_1 = true$ in $\mathcal{C}.s$ by step $f5$, so Item 5 is satisfied trivially. If $x_1 \neq r.i$ in $\mathcal{C}$, by inspection of Item 5c of I_{f2} and step $f5$, r must not be the reader satisfying Item 5c in $\mathcal{C}$. Thus, there must exist another reader satisfying Item 5c in $\mathcal{C}$ and will continue to satisfy 5c in $\mathcal{C}.s$.
I_{f2}	$f2$	$f5$	r	rest	Analogous to proof for I_{f1}
$I_{f3,f4}$	$f2$	-	w	5c	Trivially true since $x_1 = w.i$ by step $f2$
$I_{f3,f4}$	$f2$	-	w	rest	Proof analogous to that of I_{f2}
$I_{f3,f4}$	$f3$	-	w	all	Proof analogous to that of I_{f2}

$I_{f3,f4}$	$f3$, $f4$	all	r	all	Proof analogous to that of I_{f2}
I_{f5}	$f4$	-	w	4a, 5	Follows from Item 1 of I_G since $\mathcal{C} = 0$
I_{f5}	$f4$	-	w	rest	Proof analogous to that of $I_{f3,f4}$ or obvious
I_{f5}	$f5$	$r1, r3...r7$, $ra1, f1...f6$	r	all	Proof analogous to that of $I_{f3,f4}$ or obvious
I_{f5}	$f5$	$r2$	r	5	$d = D$ by step $r2$, and $Gate = \overline{D}$ in $\mathcal{C}$ by I_{f5}, so $d \neq Gate$ in $\mathcal{C}.s$, as req'd
I_{f6}	$f5$	-	w	2,3	Follows from step $f5$
I_{f6}	$f5$	-	w	5	Assume by contradiction that in $\mathcal{C}.s$, $R(PC = f6 \wedge b = Permit) \neq \phi$. Since $w.SafeID = w.b$ by step $f5$, the process(es) in $R(PC = f6 \wedge b = Permit)$ must have the same $SafeID$ as w, which contradicts Item 2 of I_G.
I_{f6}	$f5$	-	w	rest	Proof analogous to that of I_{f5} or obvious
I_{f6}	$f6$	$r1...r7, ra1$, $f1...f4, f6$	r	all	Proof analogous to that of I_{f5} or obvious Note: all CAS steps will fail because $x_1 = true$
I_{f6}	$f6$	$f5$	r	5	Assume by contradiction that in $\mathcal{C}.s$, $r \in R(PC = f6 \wedge b = Permit)$. Since $w.SafeID = w.b$ by step $f5$, r must have the same $SafeID$ as w, which contradicts Item 2 of I_G.
I_{f6}	$f6$	$f5$	r	rest	Proof analogous to that of I_{f5} or obvious
$I_{w6,wa1}$	$f1$	-	w	5c	Trivial because $x_1 = true$ from step $f1$
$I_{w6,wa1}$	$f1$	-	w	rest	Proof analogous to that of I_{f1} or obvious
$I_{w6,wa1}$	$f2$	-	w	5c	In order for the CAS to fail, $x_1 \neq w.a$ in $\mathcal{C}$, so Item 5c was satisfied by another reader in $\mathcal{C}$ by I_{f2} and will satisfy Item 5c in $\mathcal{C}.s$
$I_{w6,wa1}$	$f2$	-	w	rest	Proof analogous to that of I_{f2} or obvious

$I_{w6,wa1}$	$f3$	-	w	5c	If $x \notin PID$ in $\mathcal{C}$, Item 5 is satisfied trivially in $\mathcal{C}.s$. If $x \in PID$ in $\mathcal{C}$, by Item 5b of $I_{f3,f4}$, $Permit = w.b$, so step $f3$ must have succeeded, so w would not have stepped into $w6$
$I_{w6,wa1}$	$f3$	-	w	rest	Proof analogous to that of $I_{f3,f4}$ or obvious
$I_{w6,wa1}$	$f4$	-	w	5c	In order for w to step from $f4$ to $w6$ or $wa1$, $C \neq 0$ in $\mathcal{C}$. By I_G, $\lvert R(PC \in \{r2...r7, ra1\})\rvert \neq 0$. Since the cardinality of the set is always positive, some reader must have satisfied Item 5c of $I_{f3,f4}$ in $\mathcal{C}$ and will satisfy Item 5c of $I_{w6,wa1}$ in $\mathcal{C}.s$.
$I_{w6,wa1}$	$f4$	-	w	rest	Proof analogous to that of $I_{f3,f4}$ or obvious
$I_{w6,wa1}$	$f5$	-	w	4a	If $x_1 = true$ in $\mathcal{C}$, Item 4 holds trivially. If $x_1 \in PID$ in $\mathcal{C}$, Item 4a follows from Item 4a and Item 7 in I_{f5}.
$I_{w6,wa1}$	$f5$	-	w	4b	Follows from Item 5 of I_{f5}
$I_{w6,wa1}$	$f5$	-	w	5a	Satisfied by Item 4a of I_{f5} in $\mathcal{C}$
$I_{w6,wa1}$	$f5$	-	w	5b	Satisfied by Item 4b of I_{f5} in $\mathcal{C}$
$I_{w6,wa1}$	$f5$	-	w	5c	Since the CAS in step $f5$ failed, $x_1 \neq w.i$ in $\mathcal{C}$, so another reader must have satisfied Item 4c of I_{f5}, and will satisfy Item 5c
$I_{w6,wa1}$	$f5$	-	w	rest	Proof analogous to that of I_{f5} or obvious
$I_{w6,wa1}$	$f6$	-	w	4, 5	Trivially holds since $x_1 = true$ in $\mathcal{C}$ by Item 3 of I_{f6}
$I_{w6,wa1}$	$f6$	-	w	6a	Follows from Item 4 in I_{f6}
$I_{w6,wa1}$	$f6$	-	w	6b	Trivially holds since $Permit = w.b$ in $\mathcal{C}$ by Item 2 of I_{f6}, after step $f6$, $Permit = true$ in $\mathcal{C}.s$
$I_{w6,wa1}$	$f6$	-	w	6c	Follows from Item 6 in I_{f6}
$I_{w6,wa1}$	$f6$	-	w	rest	Proof analogous to that of I_{f6} or obvious
$I_{w6,wa1}$	$w6$	-	w	all	Proof analogous to that of I_{f2} or obvious

$I_{w6,wa1}$	$w6$, $wa1$	all	r	all	Proof analogous to that of I_{f2} or obvious
I_{wa2}	$wa1$	-	w	2	Follows from step $wa1$
I_{wa2}	$wa1$	-	w	5a, 6	Follows from Item 5, 6 of $I_{w6,wa1}$
I_{wa2}	$wa1$	-	w	rest	Proof analogous to that of $I_{w6,wa1}$ or obvious
I_{wa2}	$wa2$	all	r	all	Proof analogous to that of $I_{w6,wa1}$ or obvious Note: Since $Permit = true$ by Item 2 of I_{wa2}, the CAS in step $f6$ must fail
I_{wa3}	$wa2$	-	w	5b	$x_1 = w.a$ by step $wa2$
I_{wa3}	$wa2$	-	w	rest	Proof analogous to that of I_{wa2} or obvious
I_{wa3}	$wa3$	$r1...r4, ra1$, $f1, f2, f4...f6$	r	all	Proof analogous to that of I_{wa2} or obvious
I_{wa3}	$wa3$	$f3$	r	5b	Since $Permit = true$ by I_{wa3} in $\mathcal{C}$, PC_r cannot be in $\{f4, f5\}$ in $\mathcal{C}.s$, so Item 5b holds.
I_{wa3}	$wa3$	$f3$	r	rest	Proof analogous to that of I_{wa2} or obvious
I_{wa4}	$wa3$	-	w	4	Case 1 ($x_1 \neq w.a$ in $\mathcal{C}$): follows from Item 5b of I_{wa3} Case 2 ($x_1 = w.a$ in $\mathcal{C}$): $x_1 = nPID$ in $\mathcal{C}.s$, satisfying Item 4
I_{wa4}	$wa3$	-	w	rest	Proof analogous to that of I_{wa3} or obvious
I_{wa4}	$wa4$	all	r	all	Proof analogous to that of I_{wa3} or obvious
I_{wa5}	$wa4$	-	w	3	Follows from step $wa4$
I_{wa5}	$wa4$	-	w	5	Follows from step $wa4$ and Item 5 of I_{wa4}
I_{wa5}	$wa4$	-	w	rest	Proof analogous to that of I_{wa4} or obvious
I_{wa5}	$wa5$	$r1...r6, ra1$, $f1...f4, f6$	r	all	Proof analogous to that of I_{wa4} or obvious
I_{wa5}	$wa5$	$f5$	r	3	The CAS at Line $f5$ must fail because of Item 4 in I_{wa5} in $\mathcal{C}$
I_{wa5}	$wa5$	$f5$	r	rest	Proof analogous to that of I_{wa4} or obvious

$I_{w7,wa6}$	$wa3$	-	w	3	Follows from step $wa3$ Item 4,5:
$I_{w7,wa6}$	$wa3$	-	w	4, 5	Follows from Item 6 in I_{wa3} and the fact that $x_1 = true$ by step $wa3$
$I_{w7,wa6}$	$wa3$	-	w	rest	obvious
$I_{w7,wa6}$	$wa4$	-	w	3	Follows from step $wa4$
$I_{w7,wa6}$	$wa4$	-	w	4, 5	Follows from Item 6 in I_{wa4} and the fact that $x_1 = true$ by step $wa4$
$I_{w7,wa6}$	$wa4$	-	w	rest	obvious
$I_{w1,w2}$	$w7$, $wa6$	-	w	1	Follows from Item 1 in $I_{w7,wa6}$ and step $w7$ or step $wa6$
$I_{w1,w2}$	$w7$, $wa6$	-	w	3	Follows from Item 3, 4 in $I_{w7,wa6}$
$I_{w1,w2}$	$w7$, $wa6$	-	w	4, 5	Trivially holds since $x_1 = true$ from Item 3 of $I_{w7,wa6}$
$I_{w1,w2}$	$w7$, $wa6$	-	w	rest	obvious
$I_{w1,w2}$	$wa5$	-	w	1	Follows from Item 1 in I_{wa5} and step $wa5$
$I_{w1,w2}$	$wa5$	-	w	3	Trivially holds since $x_1 \neq true$ from Item 3 of I_{w5}
$I_{w1,w2}$	$wa5$	-	w	4	Follows from Item 3,5 in I_{wa5}
$I_{w1,w2}$	$wa5$	-	w	5	Follows from Item 4 in I_{wa5}
$I_{w1,w2}$	$wa5$	-	w	rest	obvious

7.4 Proof of Properties

Lemma 9. *(Mutual Exclusion): A reader r and the writer w cannot be in the CS together*

Proof. One can easily see from I_{w7} that there is no reader r in the CS($\mathrm{PC}_r = r7$) □

Lemma 10. *(Bounded Exit): There is an integer b such that in every run, every process completes the Exit Section in at most b of its steps.*

Proof. From inspection of Figure 2 on page 18, $w7$, $r7$, and $f1..f6$ can be executed in a bounded number of steps. Hence the lemma. □

Lemma 11. *(Concurrent Entering): There is an integer b such that, if σ is any run from a reachable configuration such that all writers are in the Remainder Section in every configuration in σ, then every read attempt in σ executes at most b steps of the Try Section before entering the CS.*

Proof. Let $b = 7$. Assume by contradiction that there is some reader r who takes more than 7 steps to enter the CS from the Try Section, then r must have stepped multiple times at $PC_r = r6$, meaning that $r.d \neq Gate$ at some point when $r6$ was executed. Since $PC_w = w1$, by inspection of Item 3, 4 of $I_{w1,w2}$, $R(PC \in \{r_3...r_6\} \wedge r.d \neq Gate) = \phi$, which is a contradiction. □

The following lemmas will be useful to prove the rest of the properties:

Lemma 12. *If at time t, a reader r is at Line $r6$ and $PC_w = w1$ then $Gate = D$ while r is at line $r6$.*

Proof. As $PC_r = r6 \wedge PC_w = w1$, from $I_{w1,w2}$, one can see that $r.d = Gate = D$. W.L.O.G., let $d = D = 1$. To prove this lemma we will show that $Gate$ is not changed while r is at Line $r6$. Say $Gate$ is set to 0, while r is still at Line $r6$. It means w executes $Gate \leftarrow 0$ ($w7$ or $wa6$) at some time after t, such that $PC_r = r6$, $D = 0$ at t. But by item 4 of I_7, $R(d = 1, PC \in \{r3...r6\}) = \phi$, which is a contradiction. □

Lemma 13. *If at time t, a reader r is at Line $r6$ and some reader is in the CS, then r is CS-enabled*

Proof. Looking through all of the invariants, in any invariant step that does *not* state $R(PC = r7) = \phi$ as a property, $R(PC = r6 \wedge d \neq Gate) = \phi$ is a property, so r must be enabled. Hence the lemma. □

Lemma 14. *(Unstoppable Reader Property) If a reader r is in the waiting room ($PC_r = r6$) at time t, then r is CS-enabled at t if any of the following holds:*

1. *if some reader is in CS at time t*
2. *$r >_{rp} w$, and w is not in the CS or Exit Section*

Proof. The first case is a mere consequence of Lemma 13. For the second case, as $r >_{rp} w$, it means that either r doorway precedes w, or some reader is in the CS when r is in the waiting room ($PC_r = r6$). In the former case it means that there is some time when $PC_r = r6$ and $PC_w = w1$, hence by Lemma 12, r should be CS-enabled. In the latter case there is some time when $PC_r = r6$ and some other reader is in the CS, hence by Lemma 13, r should be CS-enabled. □

Lemma 15. *(First-In-First-Enabled) Let r and r' be any two read attempts in a run such that r doorway precedes that of r' . If r' enters the CS before r, then r is enabled to enter the CS at the time r' enters the CS.*

Proof. FIFE follows directly from Item 1 of the Unstoppable Reader Property. □

Lemma 16. *(Reader Priority Property): If an algorithm supports only one Writer and satisfies Mutual Exclusion and Unstoppable Reader Property, then the algorithm satisfies Reader Priority property. (reproduced in verbatim from [2])*

Proof. Say r and w are read and write attempts respectively, such that $r >_{rp} w$. Assume for contradiction that w enters the CS before r, we'll call earliest such conficuration $\mathcal{C}''$. Say $\mathcal{C}'$ is the earliest configuration in which r is in the waiting room. By the definition of $r >_{rp} w$, w is not in the CS or Exit Section in $\mathcal{C}'$. This means that between $\mathcal{C}'$ and $\mathcal{C}''$, there is an intermediate configuration $\mathcal{C}$, such that w is in the Try Section and r is in the waiting room in $\mathcal{C}$. As $r >_{rp} w$ and the fact that there is only one writer in the system, by the Unstoppable Reader property, we get that r is CS-enabled in $\mathcal{C}$. Hence, r is also CS-enabled in $\mathcal{C}''$ which violates Mutal Exclusion. □

Before we prove Livelock freedom we show that no reader starves.

Lemma 17. *If a reader r is in the Try Section and no process crashes, then r eventually enters the CS.*

Proof. Suppose r stays in the Try Section forever. Then we first claim that the writer w also stays in the Try Section forever. This is true because, if r keeps taking steps it will eventually complete its doorway. Now if w ever enters the remainder section ($PC_w = w1$), by Lemma 12, r will be CS-enabled. And by assumptions of the lemma, as no process crashes, one can see both r and w will be in the Try section forever after some time t, which means$PC_r = r6$, $PC_w = w6$, and

$Permit \neq true$ forever after t. From Invariant I_{w6}, $X \neq true \implies R(PC = r6 \wedge d \neq Gate) = \phi$, so X must be $true$. However, if X is $true$, we know that $R(PC = f6 \wedge b = Permit)| = 1$, so eventually it will set $Permit = true$, which contradicts the fact that $Permit \neq true$ forever after t. □

Lemma 18. *(Livelock Freedom) If some process is in the Try Section and no process crashes, then some process enters the CS eventually.*

Proof. In the previous lemma we have shown that no reader starves. So to prove this lemma, we show that if no reader is active for all times after some time t, then the writer cannot stay in the Try Section forever. Say the writer stays in the Try Section forever after all time $t' > t$. As the writer can only stay at Line $w6$ in the Try Section forever, it means that there is some $t* > t' > t$, such that for all times after $t*$, $PC_w = w6$, $Permit \neq true$ and no readers are active.

We first claim that $X = true$ at $t*$, as $PC_w = w6$ at $t*$, by Item $5c$ of I_{w6}, one can see that some reader should be active at $t*$, which is a contradiction.

So $X \neq true \wedge PC_w = w6 \wedge Permit \neq true$. By Item $6b$ of I_6, one can again see that some reader should be active, which is a contradiction. Hence, it means that in the absence of the readers, the writer cannot stay in the Try Section forever. □

Lemma 19. *(Wait-Free Reader Abort) A reader in a busy-wait loop in the Try Section can decide to execute the Read-Abort function. We require that a process be able to complete the execution of Read-Abort in a bounded number of its own steps and subsequently enter the Remainder Section.*

Proof. From inspection of Figure 2 on page 18, $Read - Abort_i()$ clearly fulfils this property. □

Lemma 20. *(Wait-Free Writer Abort) A writer in a busy-wait loop in the Try Section can decide to execute the Write-Abort function. We require that a process be able to complete the execution of Write-Abort in a bounded number of its own steps and subsequently enter the Remainder Section.*

Proof. From inspection of Figure 2 on page 18, $Write - Abort_i()$ clearly fulfils this property. □

Lemma 21. *(Constant RMR Complexity in the Smart-Cache Model) The algorithm given in Figure 2 on page 18 has $O(1)$ RMR complexity in the Smart-Cache CC model (See Section 3.4).*

Proof. From the algorithm, we observe that all lines except Lines $w6$ and $r6$ have a constant number of steps.

Consider Line $w6$. Permit is only ever set to $\neg true$ by the writer. Since there is only one writer and readers are only capable of setting Permit to $true$, the writer can only have a cache miss at $w6$ at most 2 times (once initially, and once when Permit changes to true).

Now consider Line $r6$. W.L.O.G., say r is waiting for $Gate$ to be set to 1. By the arguments similar to the proof of Lemma 12, we know that once the $Gate$ is set to 1 while r is still at Line $r6$, it will not change to 0. Also, by the inspection of the algorithm, one can see that $Gate$ is never overwritten with the same value (either at Line $w7$ or $wa6$) . More precisely, the writer writes alternating values (1 and 0) into the $Gate$. Combining the two facts, one can see that while r is at Line $r6$, only a single write operation is performed on $Gate$. Thus, the writer will have a cache miss at $r6$ at most 2 times (once initially, and once when Gate changes). □

Recall that in the Conservative-Cache model, in any $CAS(A, u, v)$ operation where a CAS fails (and thus not changing the value of A), the system may or may not invalidate any caches containing A. In the algorithm presented in Figure 2 on page 18, this generates a worst-case RMR-complexity of $O(n_r)$, where n_r is the number of readers.

Example 22. Consider the case when there is a reader r_1 in the CS. In the meantime, a writer w enters the Try Section, executes up to Line $w6$ and sleeps. At this time, r_1 executes up to Line $f6$ and sleeps. w then decides to abort. At this time, another reader r_2 enters the CS and w enters the Try Section again. If this cycle repeats, we could have an execution where $n_r - 1$ readers are at Line $f6$ and 1 reader is in the CS. If at this point, w arrives at Line $w6$, $X \neq true$, and $Permit = w.b$ by I_{w6}. If each readers now wake up and execute Line $f6$ (and experience a CAS failure), in the Conservative-Cache model, w could potentially experience $n_r - 1$ cache misses if w wakes up in between each reader's execution of Line $f6$.

However, we can prove that our algorithm has constant amortized RMR complexity in the Conservative-Cache model by associating potential CAS failures with writer aborts.

Lemma 23. *(Constant Amortized RMR Complexity: Conservative-Cache Model) The algorithm given in Figure 2 on page 18 has $O(1)$ amortized RMR complexity in the Conservative-Cache CC model.*

Proof. We will show that each CAS failure incurred by the writer w at Line $w6$ can be attributed

to a previous call of $Write - Lock$. Since from observation of the code, w is the only process that can change X from *true* to a *non-true* value (at Line $w2$), Line $f5$ can be successfully completed at most once for every call of *Write-Lock*. (Note, however, because w can abort, not all of the *Write-Lock* calls incur the cache miss in its designated iteration). Nonetheless, each *Write-Lock* call incurs a constant amortized RMR complexity. □

8 Worst-case O(1)-RMR in Conservative Cache Model

Provided that *pid*s are contiguous, we can modify our algorithm to have constant RMR for up to R readers (where R is a constant), beyond which the algorithm will have a worst-case $O(\frac{n_r}{R})$-RMR, where n_r is the number of readers executing the algorithm. Thus, in a case where $n_r < R$, we will incur constant RMR.

To do this, we modify $Permit$ to be an array of size R and replace all instances of $Permit$ in Figure 2 on page 18 with $Permit[b\%R]$.

Thus, at any iteration of $Write - Lock$, the writer will only be waiting on the $Permit$ at index $x_2\%R$, which can have a maximum of $\frac{n_r}{R}$ readers at Line $f6$ that map to index $x_2\%R$. However, from I_G, we know that each reader will have a distinct $SafeID$, and thus, a distinct b value (from step $f5$), only one reader is poised to have a successful CAS at Line $f6$ at any given time.

This variation of the algorithm can also be proved in a similar manner using invariants. However, that is beyond the scope of this thesis and we shall be omitting this proof.

9 Generalization to Multi-Reader Multi-Writer Algorithm

Given any Abortable Mutual Exclusion algorithm Z satisfying FCFS, Bounded Exit, and Starvation Freedom, we can simply generate the abortable reader-priority multi-writer multi-reader algorithm as follows:

Let the *Write-Lock* procedure in Figure 2 on page 18 be referred to as *SW-Write-Lock* in the code below and let *Abortable-Mutex-Lock*$_i$() and *Abortable-Mutex-Unlock*$_i$() be the Lock and Unlock functions of Z:

```
procedure Write-Lock_i(){
  Abortable-Mutex-Lock_i()
  SW-Write-Lock_i()
  Abortable-Mutex-Unlock_i()
}
```

The *Read-Lock* procedure remains the same as in Figure 2 on page 18.

It is easy to observe that properties ensuring reader priority and liveness (P4, P5, RP1, RP2, A1) are fulfilled by this algorithm, as writers do not obstruct readers in the execution of Z. Moreover, P1, P2, P3, P6, and A2 are satisfied because the same properties also hold in Z.

The RMR-complexity of the resulting algorithm is the same as that of Z, since our multi-reader single-writer algorithm has constant-RMR complexity. This RMR limitation is acceptable because in the absence of readers, the abortable multi-reader multi-writer algorithm is equivalent to the abortable mutual exclusion problem. For example, we can let Z be the abortable mutual exclusion algorithm by Jayanti [10], which produces a $min(k, log(n_w))$-RMR algorithm for abortable multi-reader multi-writer algorithm (where k is the number of writers contending to enter the CS and n_w is the total number of writers). To our knowledge, this is the most efficient abortable mutual-exclusion algorithm for the CC system and it uses LL/SC objects (which can be implemented by CAS [17]).

10 Conclusion

The reader-writer problem is a variation of the mutual exclusion problem where processes are divided into readers and writers such that when a writer is accessing a shared resource, no other processes can access the resource, but may readers can access the resource at the same time.

In this thesis, we presented a constant-RMR abortable reader-priority reader-writer algorithm for Smart-Cache Cache-Coherent Systems. We also showed that this algorithm has amortized constant-RMR in Conservative-Cache Systems. We verified the model empirically using PlusCal/TLA+ to specify the algorithm and model-checking it in TLC. We then proved the algorithm using invariants. In the process of verifying this algorithm, we felt that it was inadequate to prove the algorithm without using invariants, since temporal arguments about a sufficiently long algorithm and arguments would be harder to understand and could easily be incorrect. However, it is often also laborious to manually check invariants to ensure the correctness of an invariant proof. In this thesis, an extensive attempt was made to prove the algorithm using TLAPS (TLA+ Proof System) [5]. We found TLAPS to still be in its developmental stages, as it does not reason about concepts necessary to our proof, such as set cardinality. Even besides that, perhaps due to unfamiliarity with TLAPS, we did not successfully produce a TLAPS proof of the algorithm. Regardless, we found model checking to be a powerful verification to give us confidence in our algorithm and invariants.

Moreover, we provided a constant-RMR version of the algorithm for up to R readers (where R is a constant), beyond which the RMR-complexity algorithm will be $O(\frac{n_r}{R})$, where n_r is the number of readers in the algorithm.

We also showed how to generalize the algorithm into a multi-reader multi-writer algorithm which uses (and is dependent on the time complexity of) any given abortable mutual exclusion algorithm.

11 Acknowledgments

First and foremost, I want to extend my thanks to my advisor, Professor Prasad Jayanti, and his loving family, who not only provided academic and intellectual support through one of the most challenging and exciting experiences of my time at Dartmouth, but also treated me like a member of their family and left me with a broader understanding of culture and spirituality. The time we that we spent together, be it the hours of spent verifying invariants, learning about the Puja, or experiencing the tension of the moment before India won the Cricket World Cup, are moments that will stay with me for life. Thank you for being so amazing!

I would also like to give my warmest thanks to my thesis committee members. First, to Professor Thomas Cormen, who has been my biggest influence at Dartmouth, for his continual support and encouragement to pursue computer science since my freshman fall at Dartmouth. Not only was he a great mentor, but he made me feel welcome and accepted in a major that I was very uncertain about. Truly, thank you for everything.

And also, to Professor Andrew Campbell, for being a wonderful mentor and friend who opened my eyes to the world of hacking. Never did I think I could even understand, not to mention build, a search engine, network stack, and bittorrent in mere weeks. Thank you for bringing excitement into the world of CS!

I would like to thank the members of the Dartmouth Concurrent Research Group (DCRG), Jack Bowman, Jonathan Choi, Michael Diamond, Matthew Elkherj, and Zhiyu Liu for all our great discussions and intellectual pursuit. It was wonderful getting to know all of you and going through the same experience together. I learnt so much from each one of you and our time together has truly made me even more excited about concurrent algorithms! I wish you all the best in your future endeavors.

Also, a big thank you to Ranganath Kondapally and Vibhor Bhatt for being wonderful mentors throughout my discovery of the world of algorithms and theory in computer science. I have lost count of the number of times I crowded your office hours, but had it not been for you two, I might have ended up being a biology major instead.

To my loving and caring friends who shared so much laughter and tears with me during the past four years, this would not have been possible without your love and support. Thank you so much.

And finally, to my loving parents who were behind me for the past 21 years of my life, thank you for all your love and sacrifices. There are no words to express my gratitude towards you, but truly, thank you for everything that you have done for me.

References

[1] 3 volume set of intel 64 and ia-32 architectures software developer's manuals, May 2011.

[2] Bhatt, V., and Jayanti, P. Constant RMR Solutions to Reader Writer Synchronization. Tech. rep.

[3] Brandenburg, B. B., and Anderson, J. H. Reader-writer synchronization for shared-memory multiprocessor real-time systems. In *Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 184–193.

[4] Brandenburg, B. B., and Anderson, J. H. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-time Systems 46* (2010), 25–87.

[5] Chaudhuri, K., Doligez, D., Lamport, L., and Merz, S. The tla+proof system: building a heterogeneous verification platform. In *Proceedings of the 7th International colloquium conference on Theoretical aspects of computing* (Berlin, Heidelberg, 2010), ICTAC'10, Springer-Verlag, pp. 44–44.

[6] Courtois, P. J., Heymans, F., and Parnas, D. L. Concurrent control with "readers" and "writers". *Commun. ACM 14* (October 1971), 667–668.

[7] Danek, R., and Hadzilacos, V. Local-spin group mutual exclusion algorithms. In *DISC* (2004), R. Guerraoui, Ed., vol. 3274 of *Lecture Notes in Computer Science*, Springer, pp. 71–85.

[8] Danek, R., and Hadzilacos, V. Local-spin group mutual exclusion algorithms. In *Workshop on Distributed Algorithms/International Symposium on Distributed Computing* (2004), pp. 71–85.

[9] Dijkstra, E. W. Solution of a problem in concurrent programming control. *Commun. ACM 8* (September 1965), 569–.

[10] Jayanti, P. Adaptive and efficient abortable mutual exclusion. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing* (New York, NY, USA, 2003), PODC '03, ACM, pp. 295–304.

[11] Krieger, O., Stumm, M., Unrau, R., and Hanna, J. A fair fast scalable reader-writer lock. In *In Proceedings of the 1993 International Conference on Parallel Processing* (1993), CRC Press, pp. 201–204.

[12] Lamport, L. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[13] Lamport, L. The pluscal algorithm language. In *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing* (Berlin, Heidelberg, 2009), ICTAC '09, Springer-Verlag, pp. 36–60.

[14] Lee, H. Fast local-spin abortable mutual exclusion with bounded space. In *Proceedings of the 14th international conference on Principles of distributed systems* (Berlin, Heidelberg, 2010), OPODIS'10, Springer-Verlag, pp. 364–379.

[15] Marejka, R. Atomic sparc: Using the sparc atomic instructions, Mar. 2008.

[16] Mellor-crummey, J. M., and T, M. L. S. Scalable reader-writer synchronization for shared-memory multiprocessors. In *In Proc. of the 3rd ACM SIGPLAN symposium on* (1991), pp. 106–113.

[17] Michael, M. Practical lock-free and wait-free ll/sc/vl implementations using 64-bit cas. In *Distributed Computing*, R. Guerraoui, Ed., vol. 3274 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2004, pp. 144–158.

[18] Scott, M. L., and Scherer, W. N. Scalable queue-based spin locks with timeout. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming* (New York, NY, USA, 2001), PPoPP '01, ACM, pp. 44–52.

A Appendix

A.1 Algorithm and Invariants Specified in PlusCal and TLA+

MODULE *abortable*

EXTENDS *Naturals*, *FiniteSets*, *TLC*

CONSTANTS *WriterProc*, *ReaderProc*, *nPID*, *true*

TLC complains if we use TRUE/FALSE in $X1$ and *Permit*, so let us define $0 = true$ and $1 =$ false. Thus, *WriterProc* and *ReaderProc* cannot contain $\{0, 1\}$

ASSUME $ProcType \triangleq \land WriterProc \in (Nat \setminus \{true, nPID\})$
$\land ReaderProc \subseteq (Nat \setminus \{true, nPID\})$
$\land \{WriterProc\} \cap ReaderProc = \{\}$
$\land nPID \in Nat$
$\land true \in Nat$

PlusCal Algorithm

```
–algorithm ReaderPriority{
  \ * GLOBAL VARIABLES
  variables D = FALSE,
            Gate = FALSE,
            X1 = true,
            X2 = nPID,
            Permit = true,
            C = 0;

  \ * WRITE − LOCK CODE
  process(writerprocess = WriterProc)
    variables wSafeID = WriterProc, wa, wb, prevD, currD; {
    w_ncs : while(TRUE){
    w1 : wb := X2;
    w2 : X1 := WriterProc;
         X2 := wb;
    w3 : prevD := D;
         currD := ¬prevD;
    w4 : D := currD;
    w5 : Permit := wb;

    \ * Promote(i).Not a procedure so we can reason without the stack
    wf1 : wa := X1;
          wb := X2;
          if(wa ≠ true){
    wf2 : if(X1 = wa ∧ X2 = wb){ \ * CAS(X, [wa, wb], [i, wb])
            X1 := WriterProc;
            X2 := wb;
    wf3 :       if(Permit ≠ true){
    wf4 :         if(C = 0){
    wf5 :           if(X1 = WriterProc ∧ X2 = wb){ \ * if(CAS(X, [i, wb], [true, wSafeID]))
                  X1 := true;
                  X2 := wSafeID;
                  wSafeID := wb;
    wf6 :             if(Permit = wb){ \ * CAS(Permit, wb, true)
                    Permit := true;
                  };
```

```
            };
          };
        };
      };
    };
  w6 : while(Permit ≠ true){
       either{goto w6;}
       or{
  \* WRITER ABORT SEQUENCE
  wa1 :     Permit := true;
  wa2 :     wa := X1;
        wb := X2;
  wa3 :     if(wa ≠ true){
          if(X1 = wa ∧ X2 = wb){\* CAS(X, [wa, wb], [nPID, wb])
            X1 := nPID;
            X2 := wb;
          };
  wa4 :       if(X1 ≠ true){
  wa5 :         D := prevD;
            goto w_ncs;
          };
        };
  wa6 :     goto w7;
      };
    };
  w_cs : skip;
  w7 : Gate := currD;
  }
 }
\* ********************************
  \* READ − LOCK CODE
  process(readerprocess ∈ ReaderProc)
   variables rSafeID = self, d, ra, rb;{
   r_ncs : while(TRUE){
   r1 : C := C + 1; \* atomic F & A
   r2 : d := D;
   r3 : ra := X1;     \* atomic read on X1 and X2
        rb := X2;
   r4 : if(ra ∈ ({WriterProc} ∪ ReaderProc)){
         if(X1 = ra ∧ X2 = rb){\* CAS(X, [ra, rb], [i, rb])
           X1 := self;
           X2 := rb;
         };
        };
   r5 : if(X1 = true){
   r6 :    while(Gate ≠ d){
           either{goto ra1;}or{goto r6;}; \* abort or wait
         }
```

```
      };
  r_cs : skip;
  r7 : C := C − 1; \ ∗ F & A(C, −1)
     goto rf1;

  \ ∗ READ − ABORT SEQUENCE
  ra1 : C := C − 1; \ ∗ F & A(C, −1)

  \ ∗ Promote(i).Not a procedure so we can reason without the stack
  rf1 : ra := X1;
      rb := X2;
      if(ra ≠ true){
  rf2 : if(X1 = ra ∧ X2 = rb){ \ ∗ CAS(X, [ra, rb], [i, rb])
          X1 := self;
          X2 := rb;
  rf3 :     if(Permit ≠ true){
  rf4 :       if(C = 0){
  rf5 :         if(X1 = self ∧ X2 = rb){ \ ∗ if(CAS(X, [i, rb], [true, rSafeID]))
                X1 := true;
                X2 := rSafeID;
                rSafeID := rb;
  rf6 :           if(Permit = rb){ \ ∗ CAS(Permit, rb, true)
                  Permit := true;
                };
              };
            };
          };
        };
      };
    }
   }
  }
```

BEGIN TRANSLATION

CONSTANT $defaultInitValue$

VARIABLES D, $Gate$, $X1$, $X2$, $Permit$, C, pc, $wSafeID$, wa, wb, $prevD$, $currD$, $rSafeID$, d, ra, rb

$vars \triangleq \langle D, Gate, X1, X2, Permit, C, pc, wSafeID, wa, wb, prevD, currD, rSafeID, d, ra, rb \rangle$

$ProcSet \triangleq \{WriterProc\} \cup (ReaderProc)$

$Init \triangleq$ Global variables
$\wedge D = \text{FALSE}$
$\wedge Gate = \text{FALSE}$
$\wedge X1 = true$
$\wedge X2 = nPID$
$\wedge Permit = true$
$\wedge C = 0$
Process *writerprocess*

$\land\ wSafeID = WriterProc$
$\land\ wa = defaultInitValue$
$\land\ wb = defaultInitValue$
$\land\ prevD = defaultInitValue$
$\land\ currD = defaultInitValue$
Process *readerprocess*
$\land\ rSafeID = [self \in ReaderProc \mapsto self]$
$\land\ d = [self \in ReaderProc \mapsto defaultInitValue]$
$\land\ ra = [self \in ReaderProc \mapsto defaultInitValue]$
$\land\ rb = [self \in ReaderProc \mapsto defaultInitValue]$
$\land\ pc = [self \in ProcSet \mapsto \text{CASE } self = WriterProc \to \text{“w\_ncs”}$
$\qquad\qquad\qquad\qquad\qquad\qquad \square\, self \quad \in ReaderProc \to \text{“r\_ncs”}]$

$w\_ncs \triangleq$ $\land\ pc[WriterProc] = \text{“w\_ncs”}$
$\land\ pc' = [pc \text{ EXCEPT } ![WriterProc] = \text{“w1”}]$
$\land$ UNCHANGED $\langle D, Gate, X1, X2, Permit, C, wSafeID, wa, wb, prevD, currD, rSafeID, d, ra, rb\rangle$

$w1 \triangleq$ $\land\ pc[WriterProc] = \text{“w1”}$
$\land\ wb' = X2$
$\land\ pc' = [pc \text{ EXCEPT } ![WriterProc] = \text{“w2”}]$
$\land$ UNCHANGED $\langle D, Gate, X1, X2, Permit, C, wSafeID, wa, prevD, currD, rSafeID, d, ra, rb\rangle$

$w2 \triangleq$ $\land\ pc[WriterProc] = \text{“w2”}$
$\land\ X1' = WriterProc$
$\land\ X2' = wb$
$\land\ pc' = [pc \text{ EXCEPT } ![WriterProc] = \text{“w3”}]$
$\land$ UNCHANGED $\langle D, Gate, Permit, C, wSafeID, wa, wb, prevD, currD, rSafeID, d, ra, rb\rangle$

$w3 \triangleq$ $\land\ pc[WriterProc] = \text{“w3”}$
$\land\ prevD' = D$
$\land\ currD' = (\neg prevD')$
$\land\ pc' = [pc \text{ EXCEPT } ![WriterProc] = \text{“w4”}]$
$\land$ UNCHANGED $\langle D, Gate, X1, X2, Permit, C, wSafeID, wa, wb, rSafeID, d, ra, rb\rangle$

$w4 \triangleq$ $\land\ pc[WriterProc] = \text{“w4”}$
$\land\ D' = currD$
$\land\ pc' = [pc \text{ EXCEPT } ![WriterProc] = \text{“w5”}]$
$\land$ UNCHANGED $\langle Gate, X1, X2, Permit, C, wSafeID, wa, wb, prevD, currD, rSafeID, d, ra, rb\rangle$

$w5 \triangleq$ $\land\ pc[WriterProc] = \text{“w5”}$
$\land\ Permit' = wb$
$\land\ pc' = [pc \text{ EXCEPT } ![WriterProc] = \text{“wf1”}]$

$\land$ UNCHANGED $\langle D,\ Gate,\ X1,\ X2,\ C,\ wSafeID,\ wa,\ wb,\ prevD,\ currD,$
$rSafeID,\ d,\ ra,\ rb\rangle$

$wf1 \triangleq$ $\land\ pc[WriterProc] = \text{“wf1”}$
$\land\ wa' = X1$
$\land\ wb' = X2$
$\land$ IF $wa' \neq true$
THEN $\land\ pc' = [pc$ EXCEPT $![WriterProc] = \text{“wf2”}]$
ELSE $\land\ pc' = [pc$ EXCEPT $![WriterProc] = \text{“w6”}]$
$\land$ UNCHANGED $\langle D,\ Gate,\ X1,\ X2,\ Permit,\ C,\ wSafeID,\ prevD,\ currD,$
$rSafeID,\ d,\ ra,\ rb\rangle$

$wf2 \triangleq$ $\land\ pc[WriterProc] = \text{“wf2”}$
$\land$ IF $X1 = wa \land X2 = wb$
THEN $\land\ X1' = WriterProc$
$\land\ X2' = wb$
$\land\ pc' = [pc$ EXCEPT $![WriterProc] = \text{“wf3”}]$
ELSE $\land\ pc' = [pc$ EXCEPT $![WriterProc] = \text{“w6”}]$
$\land$ UNCHANGED $\langle X1,\ X2\rangle$
$\land$ UNCHANGED $\langle D,\ Gate,\ Permit,\ C,\ wSafeID,\ wa,\ wb,\ prevD,\ currD,$
$rSafeID,\ d,\ ra,\ rb\rangle$

$wf3 \triangleq$ $\land\ pc[WriterProc] = \text{“wf3”}$
$\land$ IF $Permit \neq true$
THEN $\land\ pc' = [pc$ EXCEPT $![WriterProc] = \text{“wf4”}]$
ELSE $\land\ pc' = [pc$ EXCEPT $![WriterProc] = \text{“w6”}]$
$\land$ UNCHANGED $\langle D,\ Gate,\ X1,\ X2,\ Permit,\ C,\ wSafeID,\ wa,\ wb,\ prevD,$
$currD,\ rSafeID,\ d,\ ra,\ rb\rangle$

$wf4 \triangleq$ $\land\ pc[WriterProc] = \text{“wf4”}$
$\land$ IF $C = 0$
THEN $\land\ pc' = [pc$ EXCEPT $![WriterProc] = \text{“wf5”}]$
ELSE $\land\ pc' = [pc$ EXCEPT $![WriterProc] = \text{“w6”}]$
$\land$ UNCHANGED $\langle D,\ Gate,\ X1,\ X2,\ Permit,\ C,\ wSafeID,\ wa,\ wb,\ prevD,$
$currD,\ rSafeID,\ d,\ ra,\ rb\rangle$

$wf5 \triangleq$ $\land\ pc[WriterProc] = \text{“wf5”}$
$\land$ IF $X1 = WriterProc \land X2 = wb$
THEN $\land\ X1' = true$
$\land\ X2' = wSafeID$
$\land\ wSafeID' = wb$
$\land\ pc' = [pc$ EXCEPT $![WriterProc] = \text{“wf6”}]$
ELSE $\land\ pc' = [pc$ EXCEPT $![WriterProc] = \text{“w6”}]$
$\land$ UNCHANGED $\langle X1,\ X2,\ wSafeID\rangle$
$\land$ UNCHANGED $\langle D,\ Gate,\ Permit,\ C,\ wa,\ wb,\ prevD,\ currD,\ rSafeID,\ d,$
$ra,\ rb\rangle$

$$
\begin{array}{l}
wf6 \triangleq \ \land\, pc[WriterProc] = \text{“wf6”} \\
\quad \land \text{IF } Permit = wb \\
\qquad \text{THEN } \land\, Permit' = true \\
\qquad \text{ELSE } \land \text{TRUE} \\
\qquad\qquad\ \ \land \text{UNCHANGED } Permit \\
\quad \land\, pc' = [pc \text{ EXCEPT } ![WriterProc] = \text{“w6”}] \\
\quad \land \text{UNCHANGED } \langle D,\ Gate,\ X1,\ X2,\ C,\ wSafeID,\ wa,\ wb,\ prevD,\ currD, \\
\qquad\qquad\qquad rSafeID,\ d,\ ra,\ rb\rangle
\end{array}
$$

$$
\begin{array}{l}
w6 \triangleq \ \land\, pc[WriterProc] = \text{“w6”} \\
\quad \land \text{IF } Permit \neq true \\
\qquad \text{THEN } \land\ \lor\ \land\, pc' = [pc \text{ EXCEPT } ![WriterProc] = \text{“w6”}] \\
\qquad\qquad\qquad\ \ \lor\ \land\, pc' = [pc \text{ EXCEPT } ![WriterProc] = \text{“wa1”}] \\
\qquad \text{ELSE } \ \land\, pc' = [pc \text{ EXCEPT } ![WriterProc] = \text{“w\_cs”}] \\
\quad \land \text{UNCHANGED } \langle D,\ Gate,\ X1,\ X2,\ Permit,\ C,\ wSafeID,\ wa,\ wb,\ prevD, \\
\qquad\qquad\qquad currD,\ rSafeID,\ d,\ ra,\ rb\rangle
\end{array}
$$

$$
\begin{array}{l}
wa1 \triangleq \ \land\, pc[WriterProc] = \text{“wa1”} \\
\quad \land\, Permit' = true \\
\quad \land\, pc' = [pc \text{ EXCEPT } ![WriterProc] = \text{“wa2”}] \\
\quad \land \text{UNCHANGED } \langle D,\ Gate,\ X1,\ X2,\ C,\ wSafeID,\ wa,\ wb,\ prevD,\ currD, \\
\qquad\qquad\qquad rSafeID,\ d,\ ra,\ rb\rangle
\end{array}
$$

$$
\begin{array}{l}
wa2 \triangleq \ \land\, pc[WriterProc] = \text{“wa2”} \\
\quad \land\, wa' = X1 \\
\quad \land\, wb' = X2 \\
\quad \land\, pc' \ = [pc \text{ EXCEPT } ![WriterProc] = \text{“wa3”}] \\
\quad \land \text{UNCHANGED } \langle D,\ Gate,\ X1,\ X2,\ Permit,\ C,\ wSafeID,\ prevD,\ currD, \\
\qquad\qquad\qquad rSafeID,\ d,\ ra,\ rb\rangle
\end{array}
$$

$$
\begin{array}{l}
wa3 \triangleq \ \land\, pc[WriterProc] = \text{“wa3”} \\
\quad \land \text{IF } wa \neq true \\
\qquad \text{THEN } \land \text{IF } X1 = wa \land X2 = wb \\
\qquad\qquad\qquad\qquad \text{THEN } \land\, X1' = nPID \\
\qquad\qquad\qquad\qquad\qquad\ \ \land\, X2' = wb \\
\qquad\qquad\qquad\qquad \text{ELSE } \ \land \text{TRUE} \\
\qquad\qquad\qquad\qquad\qquad\ \ \land \text{UNCHANGED } \langle X1,\ X2\rangle \\
\qquad\qquad\qquad\ \ \land\, pc' = [pc \text{ EXCEPT } ![WriterProc] = \text{“wa4”}] \\
\qquad\quad \text{ELSE } \ \land\, pc' = [pc \text{ EXCEPT } ![WriterProc] = \text{“wa6”}] \\
\qquad\qquad\qquad\ \ \land \text{UNCHANGED } \langle X1,\ X2\rangle \\
\quad \land \text{UNCHANGED } \langle D,\ Gate,\ Permit,\ C,\ wSafeID,\ wa,\ wb,\ prevD,\ currD, \\
\qquad\qquad\qquad rSafeID,\ d,\ ra,\ rb\rangle
\end{array}
$$

$$
\begin{array}{l}
wa4 \triangleq \ \land\, pc[WriterProc] = \text{“wa4”} \\
\quad \land \text{IF } X1 \neq true \\
\qquad \text{THEN } \land\, pc' = [pc \text{ EXCEPT } ![WriterProc] = \text{“wa5”}]
\end{array}
$$

$$\begin{array}{l}
\quad \text{ELSE} \quad \land\, pc' = [pc \text{ EXCEPT } ![WriterProc] = \text{“wa6”}] \\
\land\, \text{UNCHANGED } \langle D,\, Gate,\, X1,\, X2,\, Permit,\, C,\, wSafeID,\, wa,\, wb,\, prevD, \\
\qquad\qquad currD,\, rSafeID,\, d,\, ra,\, rb\rangle
\end{array}$$

$$\begin{array}{rcl}
wa5 & \triangleq & \land\, pc[WriterProc] = \text{“wa5”} \\
& & \land\, D' = prevD \\
& & \land\, pc' = [pc \text{ EXCEPT } ![WriterProc] = \text{“w_ncs”}] \\
& & \land\, \text{UNCHANGED } \langle Gate,\, X1,\, X2,\, Permit,\, C,\, wSafeID,\, wa,\, wb,\, prevD,\, currD, \\
& & \qquad rSafeID,\, d,\, ra,\, rb\rangle
\end{array}$$

$$\begin{array}{rcl}
wa6 & \triangleq & \land\, pc[WriterProc] = \text{“wa6”} \\
& & \land\, pc' = [pc \text{ EXCEPT } ![WriterProc] = \text{“w7”}] \\
& & \land\, \text{UNCHANGED } \langle D,\, Gate,\, X1,\, X2,\, Permit,\, C,\, wSafeID,\, wa,\, wb,\, prevD, \\
& & \qquad currD,\, rSafeID,\, d,\, ra,\, rb\rangle
\end{array}$$

$$\begin{array}{rcl}
w_cs & \triangleq & \land\, pc[WriterProc] = \text{“w_cs”} \\
& & \land\, \text{TRUE} \\
& & \land\, pc' = [pc \text{ EXCEPT } ![WriterProc] = \text{“w7”}] \\
& & \land\, \text{UNCHANGED } \langle D,\, Gate,\, X1,\, X2,\, Permit,\, C,\, wSafeID,\, wa,\, wb,\, prevD, \\
& & \qquad currD,\, rSafeID,\, d,\, ra,\, rb\rangle
\end{array}$$

$$\begin{array}{rcl}
w7 & \triangleq & \land\, pc[WriterProc] = \text{“w7”} \\
& & \land\, Gate' = currD \\
& & \land\, pc' = [pc \text{ EXCEPT } ![WriterProc] = \text{“w_ncs”}] \\
& & \land\, \text{UNCHANGED } \langle D,\, X1,\, X2,\, Permit,\, C,\, wSafeID,\, wa,\, wb,\, prevD,\, currD, \\
& & \qquad rSafeID,\, d,\, ra,\, rb\rangle
\end{array}$$

$$\begin{array}{rcl}
writerprocess & \triangleq & w_ncs \lor w1 \lor w2 \lor w3 \lor w4 \lor w5 \lor wf1 \lor wf2 \lor wf3 \\
& & \quad \lor\; wf4 \lor wf5 \lor wf6 \lor w6 \lor wa1 \lor wa2 \lor wa3 \lor wa4 \\
& & \quad \lor\; wa5 \lor wa6 \lor w_cs \lor w7
\end{array}$$

$$\begin{array}{rcl}
r_ncs(self) & \triangleq & \land\, pc[self] = \text{“r_ncs”} \\
& & \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“r1”}] \\
& & \land\, \text{UNCHANGED } \langle D,\, Gate,\, X1,\, X2,\, Permit,\, C,\, wSafeID,\, wa,\, wb, \\
& & \qquad prevD,\, currD,\, rSafeID,\, d,\, ra,\, rb\rangle
\end{array}$$

$$\begin{array}{rcl}
r1(self) & \triangleq & \land\, pc[self] = \text{“r1”} \\
& & \land\, C' = C + 1 \\
& & \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“r2”}] \\
& & \land\, \text{UNCHANGED } \langle D,\, Gate,\, X1,\, X2,\, Permit,\, wSafeID,\, wa,\, wb,\, prevD, \\
& & \qquad currD,\, rSafeID,\, d,\, ra,\, rb\rangle
\end{array}$$

$$\begin{array}{rcl}
r2(self) & \triangleq & \land\, pc[self] = \text{“r2”} \\
& & \land\, d' = [d \text{ EXCEPT } ![self] = D] \\
& & \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“r3”}] \\
& & \land\, \text{UNCHANGED } \langle D,\, Gate,\, X1,\, X2,\, Permit,\, C,\, wSafeID,\, wa,\, wb,\, prevD, \\
& & \qquad currD,\, rSafeID,\, ra,\, rb\rangle
\end{array}$$

$$
\begin{array}{l}
r3(self) \triangleq \land\ pc[self] = \text{“r3”} \\
\qquad\qquad \land\ ra' = [ra \text{ EXCEPT } ![self] = X1] \\
\qquad\qquad \land\ rb' = [rb \text{ EXCEPT } ![self] = X2] \\
\qquad\qquad \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“r4”}] \\
\qquad\qquad \land \text{ UNCHANGED } \langle D,\ Gate,\ X1,\ X2,\ Permit,\ C,\ wSafeID,\ wa,\ wb,\ prevD, \\
\qquad\qquad\qquad\qquad currD,\ rSafeID,\ d\rangle \\
\\
r4(self) \triangleq \land\ pc[self] = \text{“r4”} \\
\qquad\qquad \land \text{ IF } ra[self] \in (\{WriterProc\} \cup ReaderProc) \\
\qquad\qquad\qquad \text{THEN } \land \text{ IF } X1 = ra[self] \land X2 = rb[self] \\
\qquad\qquad\qquad\qquad\qquad \text{THEN } \land\ X1' = self \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ X2' = rb[self] \\
\qquad\qquad\qquad\qquad\qquad \text{ELSE } \land \text{ TRUE} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land \text{ UNCHANGED } \langle X1,\ X2\rangle \\
\qquad\qquad\qquad \text{ELSE } \land \text{ TRUE} \\
\qquad\qquad\qquad\qquad\ \land \text{ UNCHANGED } \langle X1,\ X2\rangle \\
\qquad\qquad \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“r5”}] \\
\qquad\qquad \land \text{ UNCHANGED } \langle D,\ Gate,\ Permit,\ C,\ wSafeID,\ wa,\ wb,\ prevD,\ currD, \\
\qquad\qquad\qquad\qquad rSafeID,\ d,\ ra,\ rb\rangle \\
\\
r5(self) \triangleq \land\ pc[self] = \text{“r5”} \\
\qquad\qquad \land \text{ IF } X1 = true \\
\qquad\qquad\qquad \text{THEN } \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“r6”}] \\
\qquad\qquad\qquad \text{ELSE } \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“r\_cs”}] \\
\qquad\qquad \land \text{ UNCHANGED } \langle D,\ Gate,\ X1,\ X2,\ Permit,\ C,\ wSafeID,\ wa,\ wb,\ prevD, \\
\qquad\qquad\qquad\qquad currD,\ rSafeID,\ d,\ ra,\ rb\rangle \\
\\
r6(self) \triangleq \land\ pc[self] = \text{“r6”} \\
\qquad\qquad \land \text{ IF } Gate \neq d[self] \\
\qquad\qquad\qquad \text{THEN } \land\ \lor\ \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“ra1”}] \\
\qquad\qquad\qquad\qquad\qquad\ \lor\ \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“r6”}] \\
\qquad\qquad\qquad \text{ELSE } \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“r\_cs”}] \\
\qquad\qquad \land \text{ UNCHANGED } \langle D,\ Gate,\ X1,\ X2,\ Permit,\ C,\ wSafeID,\ wa,\ wb,\ prevD, \\
\qquad\qquad\qquad\qquad currD,\ rSafeID,\ d,\ ra,\ rb\rangle \\
\\
r\_cs(self) \triangleq \land\ pc[self] = \text{“r\_cs”} \\
\qquad\qquad \land \text{ TRUE} \\
\qquad\qquad \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“r7”}] \\
\qquad\qquad \land \text{ UNCHANGED } \langle D,\ Gate,\ X1,\ X2,\ Permit,\ C,\ wSafeID,\ wa,\ wb, \\
\qquad\qquad\qquad\qquad prevD,\ currD,\ rSafeID,\ d,\ ra,\ rb\rangle \\
\\
r7(self) \triangleq \land\ pc[self] = \text{“r7”} \\
\qquad\qquad \land\ C' = C - 1 \\
\qquad\qquad \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“rf1”}] \\
\qquad\qquad \land \text{ UNCHANGED } \langle D,\ Gate,\ X1,\ X2,\ Permit,\ wSafeID,\ wa,\ wb,\ prevD, \\
\qquad\qquad\qquad\qquad currD,\ rSafeID,\ d,\ ra,\ rb\rangle
\end{array}
$$

$$
\begin{array}{lll}
ra1(self) & \triangleq & \land\ pc[self] = \text{“ra1”} \\
& & \land\ C' = C - 1 \\
& & \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“rf1”}] \\
& & \land \text{ UNCHANGED } \langle D,\ Gate,\ X1,\ X2,\ Permit,\ wSafeID,\ wa,\ wb,\ prevD, \\
& & \qquad currD,\ rSafeID,\ d,\ ra,\ rb\rangle
\end{array}
$$

$$
\begin{array}{lll}
rf1(self) & \triangleq & \land\ pc[self] = \text{“rf1”} \\
& & \land\ ra' = [ra \text{ EXCEPT } ![self] = X1] \\
& & \land\ rb' = [rb \text{ EXCEPT } ![self] = X2] \\
& & \land \text{ IF } ra'[self] \neq true \\
& & \quad \text{THEN } \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“rf2”}] \\
& & \quad \text{ELSE } \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“r\_ncs”}] \\
& & \land \text{ UNCHANGED } \langle D,\ Gate,\ X1,\ X2,\ Permit,\ C,\ wSafeID,\ wa,\ wb, \\
& & \qquad prevD,\ currD,\ rSafeID,\ d\rangle
\end{array}
$$

$$
\begin{array}{lll}
rf2(self) & \triangleq & \land\ pc[self] = \text{“rf2”} \\
& & \land \text{ IF } X1 = ra[self] \land X2 = rb[self] \\
& & \quad \text{THEN } \land\ X1' = self \\
& & \qquad\quad \land\ X2' = rb[self] \\
& & \qquad\quad \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“rf3”}] \\
& & \quad \text{ELSE } \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“r\_ncs”}] \\
& & \qquad\quad \land \text{ UNCHANGED } \langle X1,\ X2\rangle \\
& & \land \text{ UNCHANGED } \langle D,\ Gate,\ Permit,\ C,\ wSafeID,\ wa,\ wb,\ prevD,\ currD, \\
& & \qquad rSafeID,\ d,\ ra,\ rb\rangle
\end{array}
$$

$$
\begin{array}{lll}
rf3(self) & \triangleq & \land\ pc[self] = \text{“rf3”} \\
& & \land \text{ IF } Permit \neq true \\
& & \quad \text{THEN } \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“rf4”}] \\
& & \quad \text{ELSE } \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“r\_ncs”}] \\
& & \land \text{ UNCHANGED } \langle D,\ Gate,\ X1,\ X2,\ Permit,\ C,\ wSafeID,\ wa,\ wb, \\
& & \qquad prevD,\ currD,\ rSafeID,\ d,\ ra,\ rb\rangle
\end{array}
$$

$$
\begin{array}{lll}
rf4(self) & \triangleq & \land\ pc[self] = \text{“rf4”} \\
& & \land \text{ IF } C = 0 \\
& & \quad \text{THEN } \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“rf5”}] \\
& & \quad \text{ELSE } \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“r\_ncs”}] \\
& & \land \text{ UNCHANGED } \langle D,\ Gate,\ X1,\ X2,\ Permit,\ C,\ wSafeID,\ wa,\ wb, \\
& & \qquad prevD,\ currD,\ rSafeID,\ d,\ ra,\ rb\rangle
\end{array}
$$

$$
\begin{array}{lll}
rf5(self) & \triangleq & \land\ pc[self] = \text{“rf5”} \\
& & \land \text{ IF } X1 = self \land X2 = rb[self] \\
& & \quad \text{THEN } \land\ X1' = true \\
& & \qquad\quad \land\ X2' = rSafeID[self] \\
& & \qquad\quad \land\ rSafeID' = [rSafeID \text{ EXCEPT } ![self] = rb[self]] \\
& & \qquad\quad \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“rf6”}] \\
& & \quad \text{ELSE } \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“r\_ncs”}]
\end{array}
$$

$$\land \text{ UNCHANGED } \langle X1,\ X2,\ rSafeID\rangle$$
$$\land \text{ UNCHANGED } \langle D,\ Gate,\ Permit,\ C,\ wSafeID,\ wa,\ wb,\ prevD,\ currD,\ d,\ ra,\ rb\rangle$$

$$rf6(self) \triangleq \land\ pc[self] = \text{“rf6”}$$
$$\land \text{ IF } Permit = rb[self]$$
$$\text{THEN } \land\ Permit' = true$$
$$\text{ELSE } \land \text{ TRUE}$$
$$\land \text{ UNCHANGED } Permit$$
$$\land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“r\_ncs”}]$$
$$\land \text{ UNCHANGED } \langle D,\ Gate,\ X1,\ X2,\ C,\ wSafeID,\ wa,\ wb,\ prevD,\ currD,\ rSafeID,\ d,\ ra,\ rb\rangle$$

$$readerprocess(self) \triangleq\ r\_ncs(self) \lor r1(self) \lor r2(self) \lor r3(self) \lor r4(self) \lor r5(self) \lor r6(self) \lor r\_cs(self) \lor r7(self) \lor ra1(self) \lor rf1(self) \lor rf2(self) \lor rf3(self) \lor rf4(self) \lor rf5(self) \lor rf6(self)$$

$$Next \triangleq\ writerprocess \lor (\exists\, self \in ReaderProc : readerprocess(self))$$

$$Spec \triangleq\ Init \land \Box[Next]_{vars}$$

$$Termination \triangleq\ \Diamond(\forall\, self \in ProcSet : pc[self] = \text{“Done”})$$

END TRANSLATION

List of Invariants:

$$TypeOK \triangleq\ \land\ D \in \{\text{TRUE, FALSE}\}$$
$$\land\ Gate \in \{\text{TRUE, FALSE}\}$$
$$\land\ X1 \in (ProcSet \cup \{true,\ nPID\})$$
$$\land\ X2 \in (ProcSet \cup ReaderProc \cup \{nPID\})$$
$$\land\ Permit \in (ProcSet \cup \{true,\ nPID\})$$
$$\land\ C \in Nat$$
$$\land\ wSafeID \in (ProcSet \cup \{nPID\})$$
$$\land\ \forall\, i \in ReaderProc : rSafeID[i] \in (ProcSet \cup \{nPID\})$$
$$\land\ WriterProc \in Nat \setminus \{true,\ nPID\}$$
$$\land\ ReaderProc \subseteq Nat \setminus \{true,\ nPID\}$$
$$\land\ \{WriterProc\} \cap ReaderProc = \{\}$$

$$I\_Global \triangleq\ \land\ C = Cardinality(\{i \in ReaderProc : pc[i] \in \{\text{“r2”},\ \text{“r3”},\ \text{“r4”},\ \text{“r5”},\ \text{“r6”},\ \text{“r7”},\ \text{“ra1”},\ \text{“r\_cs”}\}\})$$
$$\land\ Cardinality(\{X2\} \cup \{wSafeID\} \cup \{rSafeID[i] : i \in ReaderProc\}) = Cardinality(ProcSet) + 1$$

$$I\_w1w2 \triangleq\ pc[WriterProc] \in \{\text{“w\_ncs”},\ \text{“w1”},\ \text{“w2”}\} \Rightarrow$$
$$\land\ Gate = D$$

$$
\begin{array}{l}
\quad \land\, Permit = true \\
\quad \land\, (X1 = true) \Rightarrow (\{\} = \{j \in ReaderProc : \begin{array}{l} \land\, pc[j] \in \{\text{"r3"}, \text{"r4"}, \text{"r5"}, \text{"r6"}\} \\ \land\, d[j] \neq Gate\}) \end{array} \\
\quad \land\, (X1 \neq true) \Rightarrow (\{\} = \{j \in ReaderProc : \begin{array}{l} \land\, pc[j] = \text{"r6"} \\ \land\, d[j] \neq Gate\}) \end{array} \\
\quad \land\, \{\} = \{j \in ReaderProc : \begin{array}{l} \land\, pc[j] \in \{\text{"rf4"}, \text{"rf5"}\} \\ \land\, X1 = j\} \end{array}
\end{array}
$$

$$
\begin{array}{l}
I\_w3w4 \triangleq pc[WriterProc] \in \{\text{"w3"}, \text{"w4"}\} \Rightarrow \\
\quad \land\, Gate = D \\
\quad \land\, Permit = true \\
\quad \land\, X1 \in ProcSet \\
\quad \land\, \{\} = \{j \in ReaderProc : \begin{array}{l} \land\, pc[j] = \text{"r6"} \\ \land\, d[j] \neq Gate\} \end{array} \\
\quad \land\, \{\} = \{j \in ReaderProc : \begin{array}{l} \land\, pc[j] \in \{\text{"rf4"}, \text{"rf5"}\} \\ \land\, X1 = j\} \end{array}
\end{array}
$$

$$
\begin{array}{l}
I\_w5 \triangleq pc[WriterProc] = \text{"w5"} \Rightarrow \\
\quad \land\, Gate = \neg D \\
\quad \land\, Permit = true \\
\quad \land\, X1 \in ProcSet \\
\quad \land\, \{\} = \{j \in ReaderProc : \begin{array}{l} \land\, pc[j] = \text{"r6"} \\ \land\, d[j] \neq Gate\} \end{array} \\
\quad \land\, \{\} = \{j \in ReaderProc : \begin{array}{l} \land\, pc[j] \in \{\text{"rf4"}, \text{"rf5"}\} \\ \land\, X1 = j\} \end{array}
\end{array}
$$

$$
\begin{array}{l}
I\_f1 \triangleq pc[WriterProc] \in \{\text{"wf1"}\} \Rightarrow \\
\quad \land\, Gate = \neg D \\
\quad \land\, Permit \in \{true, nPID\} \cup ProcSet \\
\quad \land\, X1 \in ProcSet \cup \{true\} \\
\quad \land\, \{\} \neq \{j \in ReaderProc : \begin{array}{l} \land\, pc[j] \in \{\text{"rf5"}\} \\ \land\, X1 = j\} \Rightarrow \end{array} \\
\qquad \land\, \{\} = \{j \in ReaderProc : pc[j] \in \{\text{"r6"}, \text{"r7"}\}\} \\
\qquad \land\, \{\} = \{j \in ReaderProc : \begin{array}{l} \land\, pc[j] \in \{\text{"r3"}, \text{"r4"}, \text{"r5"}\} \\ \land\, d[j] = Gate\} \end{array} \\
\quad \land\, X1 \in ProcSet \Rightarrow \\
\qquad \land\, \{\} = \{j \in ReaderProc : \begin{array}{l} \land\, pc[j] = \text{"r6"} \\ \land\, d[j] \neq Gate\} \end{array} \\
\qquad \land\, Permit = wb \\
\quad \land\, X1 = true \Rightarrow \\
\qquad \land\, \{\} = \{j \in ReaderProc : \begin{array}{l} \land\, pc[j] \in \{\text{"r3"}, \text{"r4"}, \text{"r5"}, \text{"r6"}\} \\ \land\, d[j] = Gate\} \end{array} \\
\qquad \land\, Permit \neq true \Rightarrow \\
\qquad\quad 1 = Cardinality(\{j \in ReaderProc : \begin{array}{l} \land\, pc[j] = \text{"rf6"} \\ \land\, rb[j] = Permit\}) \end{array} \\
\qquad \land\, \{\} = \{j \in ReaderProc : pc[j] \in \{\text{"r7"}\}\}
\end{array}
$$

$$
\begin{array}{l}
I\_f2 \triangleq\ pc[WriterProc] \in \{\text{“wf2”}\} \Rightarrow \\
\quad \land\ Gate = \neg D \\
\quad \land\ Permit \in \{true,\ nPID\} \cup ProcSet \\
\quad \land\ X1 \in ProcSet \cup \{true\} \\
\quad \land\ \{\}\ \neq \{j \in ReaderProc :\ \land\ pc[j] \in \{\text{“rf5”}\} \\
\qquad\qquad\qquad\qquad\qquad\qquad \land\ X1 = j\} \Rightarrow \\
\qquad \land\ \{\} = \{j \in ReaderProc : pc[j] \in \{\text{“r6”},\ \text{“r7”}\}\} \\
\qquad \land\ \{\} = \{j \in ReaderProc :\ \land\ pc[j] \in \{\text{“r3”},\ \text{“r4”},\ \text{“r5”}\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ d[j] = Gate\} \\
\quad \land\ X1 \in ProcSet \Rightarrow \\
\qquad \land\ \{\} = \{j \in ReaderProc :\ \land\ pc[j] = \text{“r6”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ d[j] \neq Gate\} \\
\qquad \land\ Permit = wb \\
\qquad \land\ X1 \neq wa \Rightarrow\ \lor\ \{\} \neq \{j \in ReaderProc : \\
\qquad\qquad\qquad\qquad\qquad pc[j] \in \{\text{“r2”},\ \text{“r3”},\ \text{“r4”},\ \text{“r5”},\ \text{“r6”},\ \text{“r\_cs”},\ \text{“r7”},\ \text{“ra1”},\ \text{“rf1”}\}\} \\
\qquad\qquad\qquad\qquad\qquad \cup \\
\qquad\qquad\qquad\qquad\qquad \{j \in ReaderProc :\ \land\ pc[j] = \text{“rf2”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ X1 = ra[j]\} \\
\qquad\qquad\qquad\quad \lor\ \{\} \neq \{j \in ReaderProc :\ \land\ pc[j] \in \{\text{“rf3”},\ \text{“rf4”},\ \text{“rf5”}\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ X1 = j\} \\
\quad \land\ X1 = true \Rightarrow \\
\qquad \land\ \{\} = \{j \in ReaderProc :\ \land\ pc[j] \in \{\text{“r3”},\ \text{“r4”},\ \text{“r5”},\ \text{“r6”}\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ d[j] = Gate\} \\
\qquad \land\ Permit \neq true \Rightarrow \\
\qquad\quad 1 = Cardinality(\{j \in ReaderProc :\ \land\ pc[j] = \text{“rf6”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ rb[j] = Permit\}) \\
\qquad \land\ \{\} = \{j \in ReaderProc : pc[j] \in \{\text{“r7”}\}\}
\end{array}
$$

$$
\begin{array}{l}
I\_f3f4 \triangleq\ pc[WriterProc] \in \{\text{“wf3”},\ \text{“wf4”}\} \Rightarrow \\
\quad \land\ Gate = \neg D \\
\quad \land\ Permit \in \{true,\ nPID\} \cup ProcSet \\
\quad \land\ X1 \in ProcSet \cup \{true\} \\
\quad \land\ \{\}\ \neq \{j \in ReaderProc :\ \land\ pc[j] \in \{\text{“rf5”}\} \\
\qquad\qquad\qquad\qquad\qquad\qquad \land\ X1 = j\} \Rightarrow \\
\qquad \land\ \{\} = \{j \in ReaderProc : pc[j] \in \{\text{“r6”},\ \text{“r7”}\}\} \\
\qquad \land\ \{\} = \{j \in ReaderProc :\ \land\ pc[j] \in \{\text{“r3”},\ \text{“r4”},\ \text{“r5”}\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ d[j] = Gate\} \\
\quad \land\ X1 \in ProcSet \Rightarrow \\
\qquad \land\ \{\} = \{j \in ReaderProc :\ \land\ pc[j] = \text{“r6”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ d[j] \neq Gate\} \\
\qquad \land\ Permit = wb \\
\qquad \land\ X1 \neq WriterProc \Rightarrow (\ \lor\ \{\} \neq \{j \in ReaderProc : \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad pc[j] \in \{\text{“r2”},\ \text{“r3”},\ \text{“r4”},\ \text{“r5”},\ \text{“r6”},\ \text{“r\_cs”},\ \text{“r7”},\ \text{“ra1”},\ \text{“rf1”}\}\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cup \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{j \in ReaderProc :\ \land\ pc[j] = \text{“rf2”}
\end{array}
$$

$$\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge X1 = ra[j]\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \{\} \neq \{j \in ReaderProc : \wedge pc[j] \in \{\text{“rf3”}, \text{“rf4”}, \text{“rf5”}\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge X1 = j\}) \\
\quad \wedge X1 = true \Rightarrow \\
\quad\quad \wedge\ \{\} = \{j \in ReaderProc : \wedge pc[j] \in \{\text{“r3”}, \text{“r4”}, \text{“r5”}, \text{“r6”}\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge d[j] = Gate\} \\
\quad\quad \wedge\ Permit \neq true \Rightarrow \\
\quad\quad\quad 1 = Cardinality(\{j \in ReaderProc : \wedge pc[j] = \text{“rf6”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge rb[j] = Permit\}) \\
\quad\quad \wedge\ \{\} = \{j \in ReaderProc : pc[j] \in \{\text{“r7”}\}\}
\end{array}$$

$$\begin{array}{l}
I_f5 \triangleq\ pc[WriterProc] = \text{“wf5”} \Rightarrow \\
\quad \wedge Gate = \neg D \\
\quad \wedge Permit \in \{true, nPID\} \cup ProcSet \\
\quad \wedge X1 \in ProcSet \cup \{true\} \\
\quad \wedge X1 \in ProcSet \Rightarrow \\
\quad\quad \wedge\ \{\} = \{j \in ReaderProc : pc[j] = \text{“r6”}\} \\
\quad\quad \wedge\ Permit = wb \\
\quad\quad \wedge\ X1 \neq WriterProc \Rightarrow (\\
\quad\quad\quad \vee\ \{\} \neq \{j \in ReaderProc : pc[j] \in \{\text{“r2”}, \text{“r3”}, \text{“r4”}, \text{“r5”}, \text{“r6”}, \text{“r_cs”}, \text{“r7”}, \text{“ra1”}, \text{“rf1”}\}\} \\
\qquad\qquad\qquad\quad \cup \\
\qquad\qquad\qquad\quad \{j \in ReaderProc : \wedge pc[j] = \text{“rf2”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge ra[j] = X1\} \\
\quad\quad\quad \vee\ \{\} \neq \{j \in ReaderProc : \wedge pc[j] \in \{\text{“rf3”}, \text{“rf4”}, \text{“rf5”}\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge j = X1\}) \\
\quad \wedge \{\}\ = \{j \in ReaderProc : \wedge pc[j] \in \{\text{“r3”}, \text{“r4”}, \text{“r5”}, \text{“r6”}\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge d[j] = Gate\} \\
\quad \wedge X1 = true \Rightarrow \\
\quad\quad \wedge\ Permit \neq true \Rightarrow \\
\quad\quad\quad 1 = Cardinality(\{j \in ReaderProc : \wedge pc[j] = \text{“rf6”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge rb[j] = Permit\}) \\
\quad\quad \wedge\ \{\} = \{j \in ReaderProc : pc[j] \in \{\text{“r7”}\}\}
\end{array}$$

$$\begin{array}{l}
I_f6 \triangleq\ pc[WriterProc] = \text{“wf6”} \Rightarrow \\
\quad \wedge Gate = \neg D \\
\quad \wedge Permit = wb \\
\quad \wedge X1 = true \\
\quad \wedge \{\}\ = \{j \in ReaderProc : \wedge pc[j] \in \{\text{“r3”}, \text{“r4”}, \text{“r5”}, \text{“r6”}\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge d[j] = Gate\} \\
\quad \wedge \{\}\ = \{j \in ReaderProc : \wedge pc[j] = \text{“rf6”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge rb[j] = Permit\} \\
\quad \wedge \{\}\ = \{j \in ReaderProc : pc[j] \in \{\text{“r7”}\}\}
\end{array}$$

$$\begin{array}{l}
I_w6wa1 \triangleq\ pc[WriterProc] \in \{\text{“w6”}, \text{“wa1”}\} \Rightarrow \\
\quad \wedge Gate = \neg D \\
\quad \wedge Permit \in \{true, nPID\} \cup ProcSet
\end{array}$$

$\land X1 \in ProcSet \cup \{true\}$
$\land \{\} \neq \{j \in ReaderProc : \land pc[j] \in \{\text{“rf5”}\}$
$\land X1 = j\} \Rightarrow$
$\land \{\} = \{j \in ReaderProc : pc[j] \in \{\text{“r6”}, \text{“r7”}\}\}$
$\land \{\} = \{j \in ReaderProc : \land pc[j] \in \{\text{“r3”}, \text{“r4”}, \text{“r5”}\}$
$\land d[j] = Gate\}$
$\land X1 \in ProcSet \Rightarrow$
$\land \{\} = \{j \in ReaderProc : \land pc[j] = \text{“r6”}$
$\land d[j] \neq Gate\}$
$\land Permit = wb$
$\land \lor \{\} \neq \{j \in ReaderProc : pc[j] \in \{\text{“r2”}, \text{“r3”}, \text{“r4”}, \text{“r5”}, \text{“r6”}, \text{“r\_cs”}, \text{“r7”}, \text{“ra1”}, \text{“rf1”}\}\}$
$\cup$
$\{j \in ReaderProc : \land pc[j] = \text{“rf2”}$
$\land ra[j] = X1\}$
$\lor \{\} \neq \{j \in ReaderProc : \land pc[j] \in \{\text{“rf3”}, \text{“rf4”}, \text{“rf5”}\}$
$\land j = X1\}$
$\land X1 = true \Rightarrow$
$\land \{\} = \{j \in ReaderProc : \land pc[j] \in \{\text{“r3”}, \text{“r4”}, \text{“r5”}, \text{“r6”}\}$
$\land d[j] = Gate\}$
$\land Permit \neq true \Rightarrow$
$1 = Cardinality(\{j \in ReaderProc : \land pc[j] = \text{“rf6”}$
$\land rb[j] = Permit\})$
$\land \{\} = \{j \in ReaderProc : pc[j] \in \{\text{“r7”}\}\}$

$I\_w7wa6 \triangleq pc[WriterProc] \in \{\text{“w7”}, \text{“wa6”}\} \Rightarrow$
$\land Gate = \neg D$
$\land Permit = true$
$\land X1 = true$
$\land \{\} = \{j \in ReaderProc : \land pc[j] \in \{\text{“r3”}, \text{“r4”}, \text{“r5”}, \text{“r6”}\}$
$\land d[j] = Gate\}$
$\land \{\} = \{j \in ReaderProc : pc[j] \in \{\text{“r7”}\}\}$

$I\_wa2 \triangleq pc[WriterProc] \in \{\text{“wa2”}\} \Rightarrow$
$\land Gate = \neg D$
$\land Permit = true$
$\land X1 \in ProcSet \cup \{true\}$
$\land \{\} \neq \{j \in ReaderProc : \land pc[j] \in \{\text{“rf5”}\}$
$\land X1 = j\} \Rightarrow$
$\land \{\} = \{j \in ReaderProc : pc[j] \in \{\text{“r6”}, \text{“r7”}\}\}$
$\land \{\} = \{j \in ReaderProc : \land pc[j] \in \{\text{“r3”}, \text{“r4”}, \text{“r5”}\}$
$\land d[j] = Gate\}$
$\land X1 \in ProcSet \Rightarrow$
$\land \{\} = \{j \in ReaderProc : \land pc[j] = \text{“r6”}$
$\land d[j] \neq Gate\}$
$\land X1 = true \Rightarrow$

$\land\ \{\} = \{j \in ReaderProc :\ \land pc[j] \in \{\text{“r3”},\ \text{“r4”},\ \text{“r5”},\ \text{“r6”}\}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land d[j] = Gate\}$
$\land\ \{\} = \{j \in ReaderProc : pc[j] \in \{\text{“r7”}\}\}$

$I\_wa3 \triangleq\ pc[WriterProc] \in \{\text{“wa3”}\} \Rightarrow$
$\land\ Gate = \neg D$
$\land\ Permit = true$
$\land\ X1 \in ProcSet \cup \{true\}$
$\land\ \{\}\ \neq \{j \in ReaderProc :\ \land pc[j] \in \{\text{“rf5”}\}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land X1 = j\} \Rightarrow$
$\qquad\land \{\} = \{j \in ReaderProc : pc[j] \in \{\text{“r6”},\ \text{“r7”}\}\}$
$\qquad\land \{\} = \{j \in ReaderProc :\ \land pc[j] \in \{\text{“r3”},\ \text{“r4”},\ \text{“r5”}\}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land d[j] = Gate\}$
$\land\ X1 \in ProcSet \Rightarrow$
$\qquad\land\ \{\}\ = \{j \in ReaderProc :\ \land pc[j] = \text{“r6”}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land d[j] \neq Gate\}$
$\qquad\land\ X1 \neq wa \Rightarrow \{\} = \{j \in ReaderProc :\ \land pc[j] \in \{\text{“f4”},\ \text{“f5”}\}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land X1 = j\}$
$\land\ X1 = true \Rightarrow$
$\qquad\land\ \{\} = \{j \in ReaderProc :\ \land pc[j] \in \{\text{“r3”},\ \text{“r4”},\ \text{“r5”},\ \text{“r6”}\}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land d[j] = Gate\}$
$\qquad\land\ \{\} = \{j \in ReaderProc : pc[j] \in \{\text{“r7”}\}\}$

$I\_wa4 \triangleq\ pc[WriterProc] \in \{\text{“wa4”}\} \Rightarrow$
$\land\ Gate = \neg D$
$\land\ Permit = true$
$\land\ X1 \in ProcSet \cup \{true,\ nPID\}$
$\land\ \{\}\ = \{j \in ReaderProc :\ \land pc[j] \in \{\text{“rf4”},\ \text{“rf5”}\}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land X1 = j\}$
$\land\ X1 \neq true \Rightarrow$
$\qquad\land\ \{\} = \{j \in ReaderProc :\ \land pc[j] = \text{“r6”}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land d[j] \neq Gate\}$
$\land\ X1 = true \Rightarrow$
$\qquad\land\ \{\} = \{j \in ReaderProc :\ \land pc[j] \in \{\text{“r3”},\ \text{“r4”},\ \text{“r5”},\ \text{“r6”}\}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land d[j] = Gate\}$
$\qquad\land\ \{\} = \{j \in ReaderProc : pc[j] \in \{\text{“r7”}\}\}$

$I\_wa5 \triangleq\ pc[WriterProc] \in \{\text{“wa5”}\} \Rightarrow$
$\land\ Gate = \neg D$
$\land\ Permit = true$
$\land\ X1 \neq true$
$\land\ \{\}\ = \{j \in ReaderProc :\ \land pc[j] \in \{\text{“rf4”},\ \text{“rf5”}\}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land X1 = j\}$
$\land\ \{\}\ = \{j \in ReaderProc :\ \land pc[j] = \text{“r6”}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land d[j] \neq Gate\}$

$$
\begin{aligned}
\mathit{InvAll} \triangleq\ & \land\ \mathit{TypeOK} \\
& \land\ I\_\mathit{Global} \\
& \land\ I\_w1w2 \\
& \land\ I\_w3w4 \\
& \land\ I\_w5 \\
& \land\ I\_f1 \\
& \land\ I\_f2 \\
& \land\ I\_f3f4 \\
& \land\ I\_f5 \\
& \land\ I\_f6 \\
& \land\ I\_w6wa1 \\
& \land\ I\_w7wa6 \\
& \land\ I\_wa2 \\
& \land\ I\_wa3 \\
& \land\ I\_wa4 \\
& \land\ I\_wa5
\end{aligned}
$$

* Modification History
* Last modified *Wed* May 25 12:59:22 *EDT* 2011 by *NancyZheng*
* Last modified *Tue* May 10 13:46:19 *EDT* 2011 by *Nancy*
* Created Sun *Feb* 27 22:17:05 *EST* 2011 by *Nancy*